Ordenação

QuickSort e HeapSort

Profª. Barbara Quintela
Prof. Jose J. Camata
Prof. Marcelo Caniato

barbara@ice.ufjf.br
camata@ice.ufjf.br
marcelo.caniato@ice.ufjf.br

QuickSort - Introdução

- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna **mais rápido** que se conhece para uma ampla variedade de situações.
- Aplica-se o paradigma de **divisão e conquista**:
  - **Divisão**: particionar a sequência de entrada L[*p,...,r*] em dois subarranjos L[*p,...q-1*] e L[*q+1,...,r*] tais que:
    - L[*p,...q-1*] contém elementos menores ou igual L[q]
    - L[*q+1,...,r*] contém elementos maiores ou igual a L[q]
  - **Conquista:** Ordenar os dois subarranjos por chamadas recursivas.
  - **Combinação:** Nada a ser feito, subarranjos retornam ordenados na etapa anterior.

QuickSort - Particionamento

- A parte mais delicada do método é o processo de partição.
- Particionamento:
  - Dados uma sequência de entrada L[p,...,r] e um elemento de L denominado pivô:
  - A seqûencia L será particionado em duas partes:
    - A parte a esquerda com chaves menores ou iguais ao pivô.
    - A parte a direita com chaves maiores ou iguais ao pivô.

QuickSort - Particionamento

- A parte mais delicada do método é o processo de partição.
- Algoritmo para o particionamento:
  - Escolha arbitrariamente um pivô.
  - Percorra o vetor da esquerda para direita enquanto L[i] < pivô.
  - Percorra o vetor da direita para esquerda enquanto L[j] > pivo.
  - Troque L[i] com L[j].
  - Continue este processo até os apontadores i e j se cruzarem.

O particionamento Ilustrado...

0	1	2	3	4	5
O	R	D	E	N	A

(1) **Escolha pivô:** Vamos escolher o elemento localizado no centro do vetor

(2) m = (0+5) div 2 = 2

(3) pivô = D

O particionamento Ilustrado...

0	1	2	3	4	5
O	R	D	E	N	A

i

j

PIVÔ: D

O particionamento Ilustrado...

0	1	2	3	4	5
O	R	D	E	N	A

i

O < D? **(F)**

j

A > D? **(F)**

PIVÔ: D	troca A com O

O particionamento Ilustrado...

0	1	2	3	4	5
A	R	D	E	N	O

i

j

PIVÔ: D — atualiza i e j

O particionamento Ilustrado...

0	1	2	3	4	5
A	R	D	E	N	O

i

j

R < D? **(F)**

N > D? **(V)**

PIVÔ: D	atualiza j

O particionamento Ilustrado...

0	1	2	3	4	5
A	R	D	E	N	O

i

j

R < D? **(F)**

E > D? **(V)**

PIVÔ: D	atualiza j

O particionamento Ilustrado...

0	1	2	3	4	5
A	R	D	E	N	O

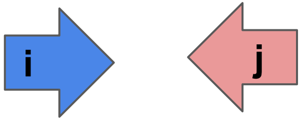
i j

R < D? **(F)** D > D? **(F)**

PIVÔ: D	troca R com D

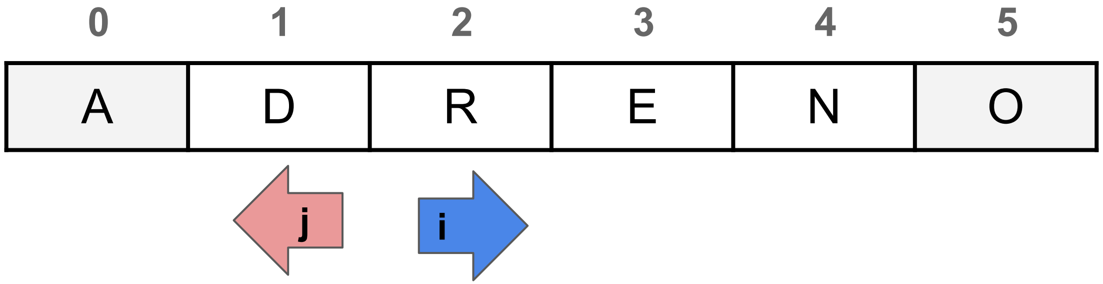
O particionamento Ilustrado...

0	1	2	3	4	5
A	D	R	E	N	O



PIVÔ: D	atualiza i e j

O particionamento Ilustrado...



PIVÔ: D	Interrompre processo e retorne j

Algoritmo de Particionamento

```
particionamento(L:vetor, lo: int, hi: int)
  Escolha PIVÔ := A[ floor((hi + lo) / 2) ]
  i = lo - 1
  j = hi + 1
  Faça
    Faça i ← i + 1 Enquanto (L[i] < PIVÔ)
    Faça j ← j - 1 Enquanto (L[j] > PIVÔ)
    Se (i >= j ) Então return j
    Troca(L[i],L[j])
  Enquanto (True) ;
```

Particionamento: Análise

- Roda em O(n)
- Número de comparações é independente de como os dados estão arranjados
- Número de trocas, por outro lado, é dependente do arranjo das informações:
  - Se a ordem está invertida e o pivô divide exatamente em dois conjuntos de mesmo tamanho temos *n/2* trocas
  - No caso de dados randômico há um pouco menos de n/2 trocas

Quicksort - Algoritmo

- Após o particionamento do vetor temos dois grupo internos:
  - Do lado esquerdo do pivô com valores menores
  - Do lado direito do pivô com valores maiores.
  - O pivô já está na posição correta no vetor ordenado
- Se ordenarmos cada um desses grupos, temos no final um vetor totalmente ordenado.
- Como ordená-los?
  - Utilizando as chamadas recursivas para particionar cada um desses lados.
  - Se o subvetor contém somente um elemento, este é o critério de parada para as chamadas recursivas

QuickSort - Algoritmo

vetor não particionado

O	R	D	E	N	A

vetor após particionamento

A	D	R	E	N	**O**

Será ordenado após primeira chamada recursiva do QuickSort

Será ordenado após segunda chamada recursiva do QuickSort

QuickSort- Algoritmo

```
QuickSort(L:vetor, lo: int, hi: int)
Se (lo < hi) Então
    p = particionamento(L,p,r);
    QuickSort(L, lo , p);
    QuickSort(L, p+1, hi);
Fim-Se
```

QuickSort: Análise

- Pior caso: $O(n^2)$
  - O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
  - Isto faz com que o quicksort seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada.
- Melhor Caso: $O(n \log n)$
- Caso médio: $O(n \log n)$

Escolha do Pivô

- A escolha do pivô pode influenciar no desempenho do método
- O pivô deve ser algum dos valores que compõem a sequência de entrada
- O pivô pode ser escolhido aleatoriamente.

Escolha do Pivô

- Analisando algumas opções:
  - Escolher o **primeiro ou o último elemento** do vetor?
    - **prós:** Simples de codificar, rápido de calcular
    - **contra**: Se o vetor estiver quase ordenado ou em ordem reversa, o quicksort pode degradar para $O(n^2)$.

Escolha do Pivô

- Analisando algumas opções:
  - Escolher o **elemento localizado no meio** do vetor?
    - **prós:** Simples de codificar, rápido de calcular mas um pouco mais lento que os métodos anteriores
    - **contra**: Se o vetor estiver quase ordenado ou em ordem reversa, o quicksort pode degradar para $O(n^2)$.

Escolha do Pivô

- Analisando algumas opções:
  - Escolher o escolha o **valor mediano** entre o primeiro, o último e o elemento do meio do vetor?
    - **prós:** Simples de codificar, razoavelmente rápido de calcular porém um pouco mais lento que os métodos anteriores
    - **contra**: Ainda pode degradar para $O(n^2)$. Bastante fácil para alguém construir uma vetor que irá degradar para $O(n^2)$.

Escolha do Pivô

- Analisando algumas opções:
  - **Escolha o pivô aleatoriamente** (usando uma função aleatória personalizada):
    - **prós:** Muito mais difícil para alguém construir um vetor que irá degradar para $O(n^2)$, se não souberem como se está escolhendo os números aleatórios.
    - **contra**: Pode ser complicado de codificar. A seleção de um pivô aleatório é bastante lenta. Ainda é teoricamente possível que o método degrada para $O(n^2)$.

Escolha do Pivô

- Analisando algumas opções:
  - Escolha **mediana das medianas** para selecionar um pivô
    - **prós:** O pivô é garantido como bom. Quicksort agora é O (*nlogn*) no pior dos casos!
    - **contra**: código mais complexo

Qual método de escolha de pivô devo usar?

- Se é improvável que os dados já estejam ordenados e é aceitável O(n^2) nos casos raros em que o vetor está ordenado, use o elemento mais à esquerda ou à direita
- Se houver uma chance razoável de que seus dados estejam ordenados, use o elemento do meio ou a mediana de três
- Se você estiver um pouco preocupado com usuários mal-intencionados, dando a você vetores ruins para ordenar, use pivôs aleatórios
- Se precisar garantir que o quicksort seja O ($n \log n$), use a mediana das medianas.

QuickSort: Resumo

- **Vantagens:**
  - É extremamente eficiente para ordenar arquivos de dados.
  - Necessita de apenas uma pequena pilha como memória auxiliar.
  - Requer cerca de *n log n* comparações em média para ordenar n itens.
- **Desvantagens:**
  - Tem um pior caso $O(n^2)$ comparações.
  - Sua implementação é muito delicada:
    - Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
  - O método **não é estável.**

Próximo vídeo:

HeapSort

HeapSort

HeapSort: Introdução

- Algoritmo criado por John Williams (1964)
- Possui o mesmo princípio de funcionamento da ordenação por seleção.
- Algoritmo:
  a. Selecione o menor item do vetor.
  b. Troque-o com o item da primeira posição do vetor.
  c. Repita estas operações com os $n - 1$ itens restantes, depois com os $n - 2$ itens, e assim sucessivamente.
- O custo para encontrar o menor item entre n itens é $n - 1$ comparações.
- Custo reduzido utilizando uma heap.

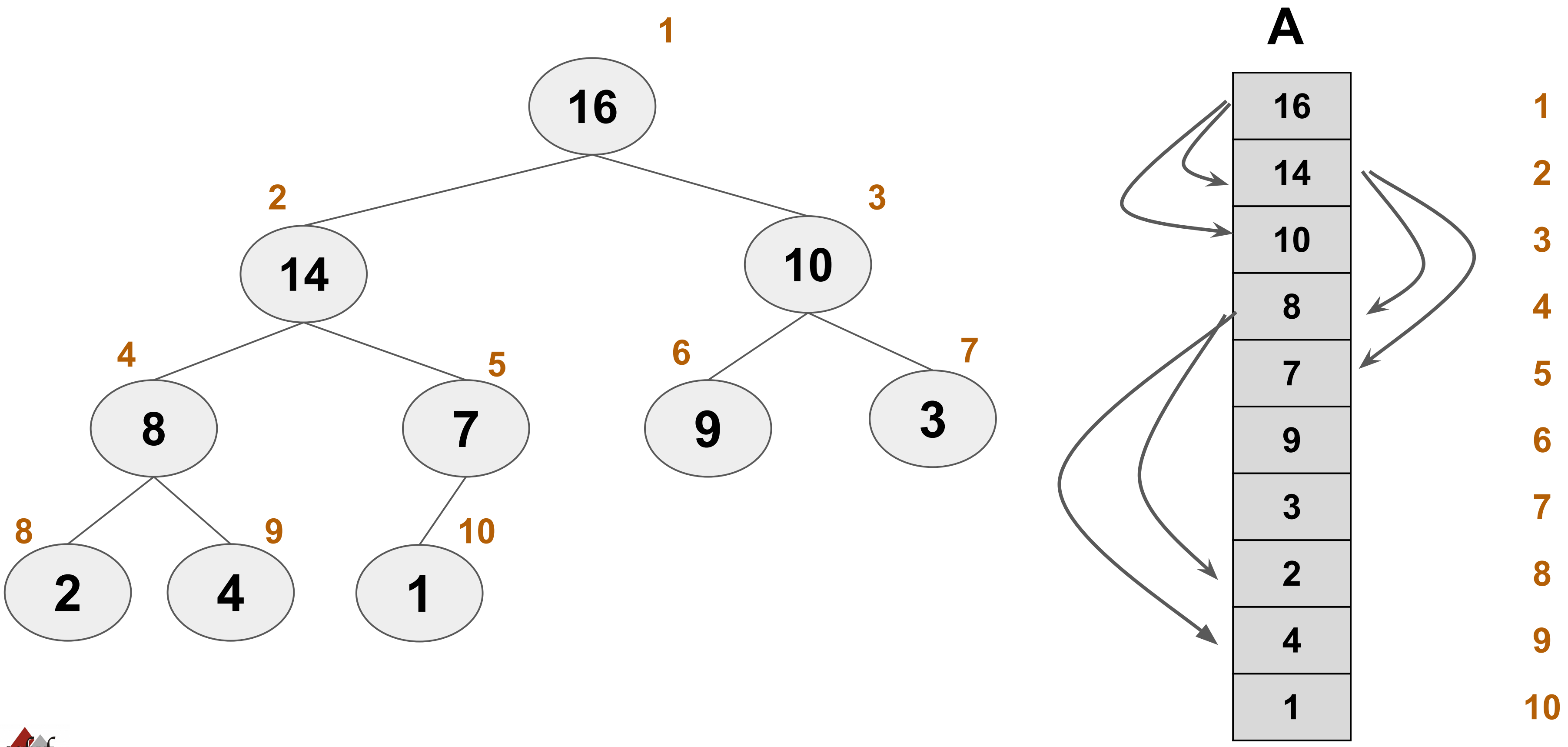
Estrutura de Dados: Heap

- A estrutura de dados heap é um objeto arranjo A que pode ser visto como uma árvore binária quase completa.
- Possui as seguinte propriedades:
  - *Cada nó da árvore corresponde a um elemento do arranjo A.*
  - *A árvore está sempre completamente preenchida em todos os níveis, exceto o último, que é preenchido a partir da esquerda.*
  - *O valor de cada nó não é menor que os valores armazenados em cada filho.*

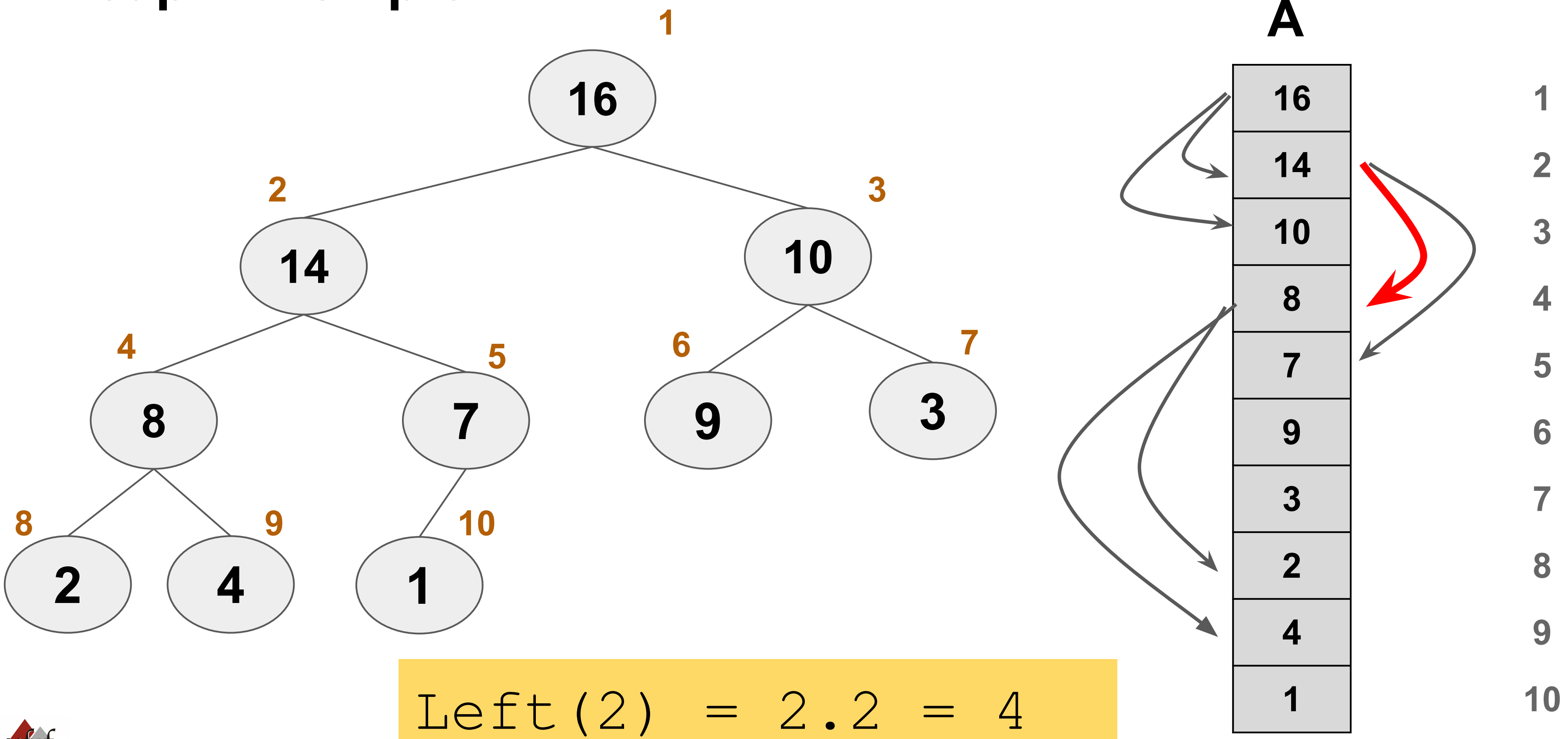
Heap

- Observações:
  - Elementos no heap não estão perfeitamente ordenados.
  - Tenta-se evitar a utilização real de uma árvore usando ponteiros
- Dado um arranjo A[1…n], verifique que, caso o arranjo representa uma árvore, então:
  - Raiz da árvore está em A[1].
  - Dado um índice *i*:
    - O pai de uma índice *i* é *i div 2*
      - Notação: `Parent(i)`
    - O filho esquerdo do índice *i* é *2i*
      - Notação: `Left(i)`
    - O filho direito do índice *i* é *2i+1*
      - Notação: `Right(i)`

Heap - Exemplo

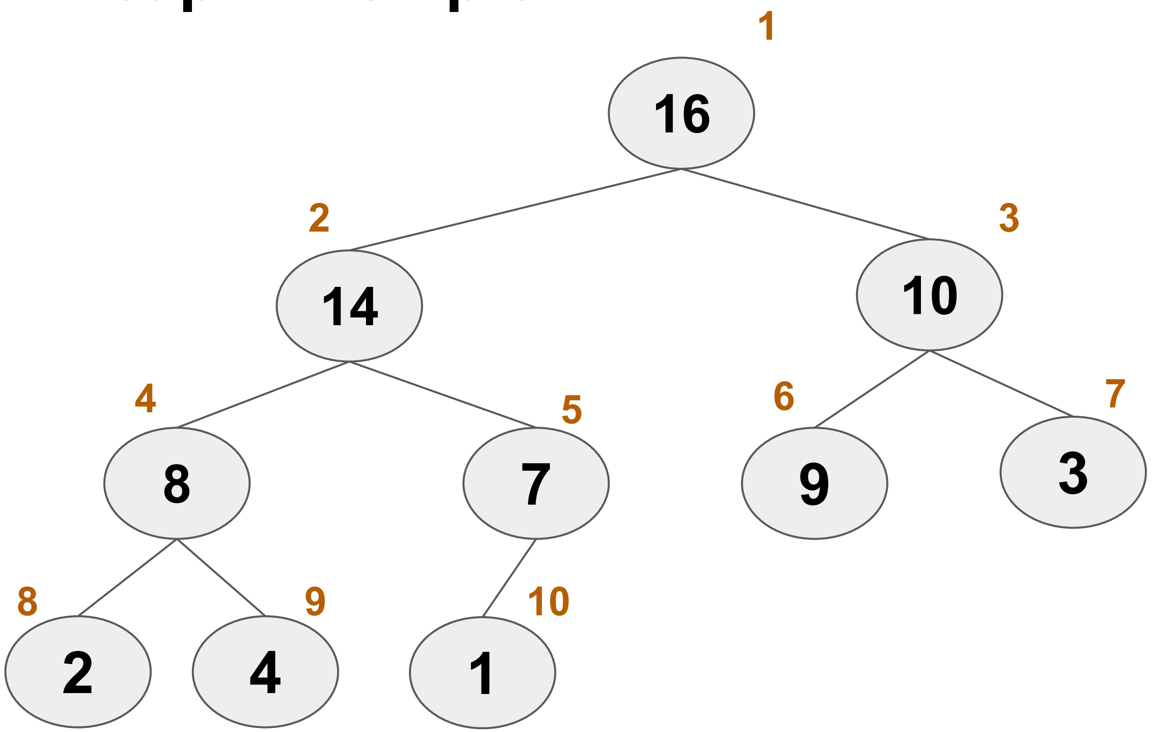


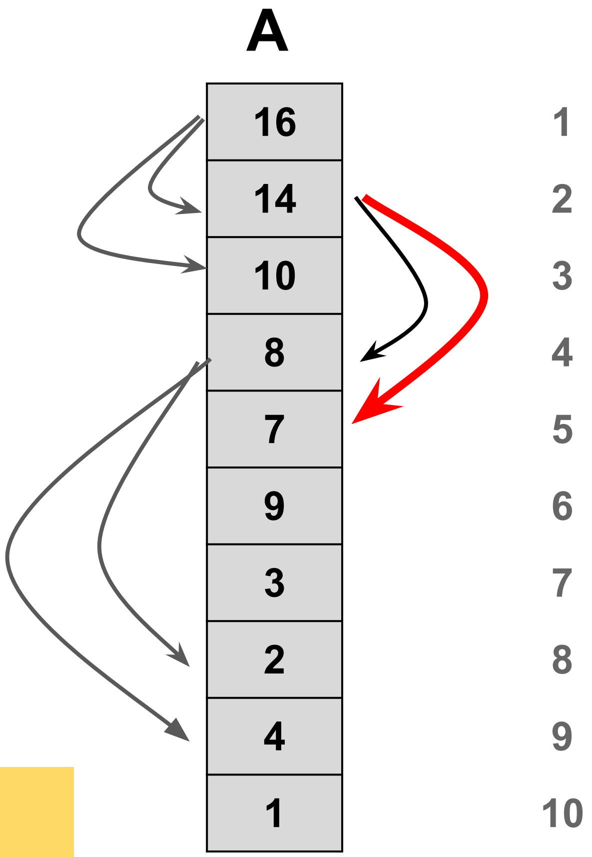
Heap - Exemplo



`Left(2) = 2.2 = 4`

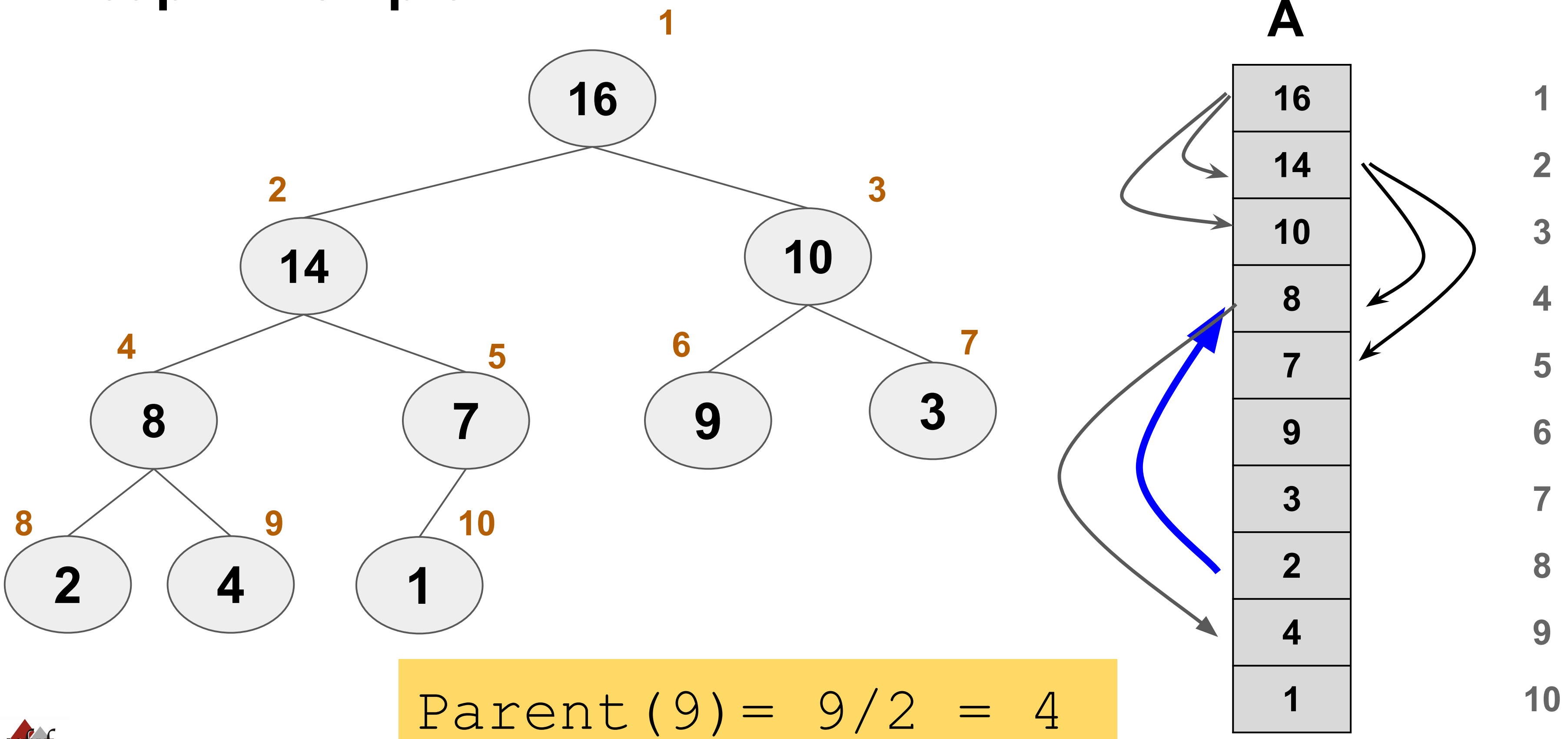
Heap - Exemplo






```
Right(2)=2.2+1=5
```

Heap - Exemplo



`Parent(9)= 9/2 = 4`

Heap de Máximo e Heap de Mínimo

Existem dois tipos de heaps binários:

- *Heap de Máximo:* Para todo nó *i* exceto a raiz

  A[Parent(i)] >= A[i]

  - O valor de um nó é no máximo o valor do nó pai.
  - O maior valor está na raiz da heap
- *Heap de Mínimo:* Para todo nó *i* exceto a raiz

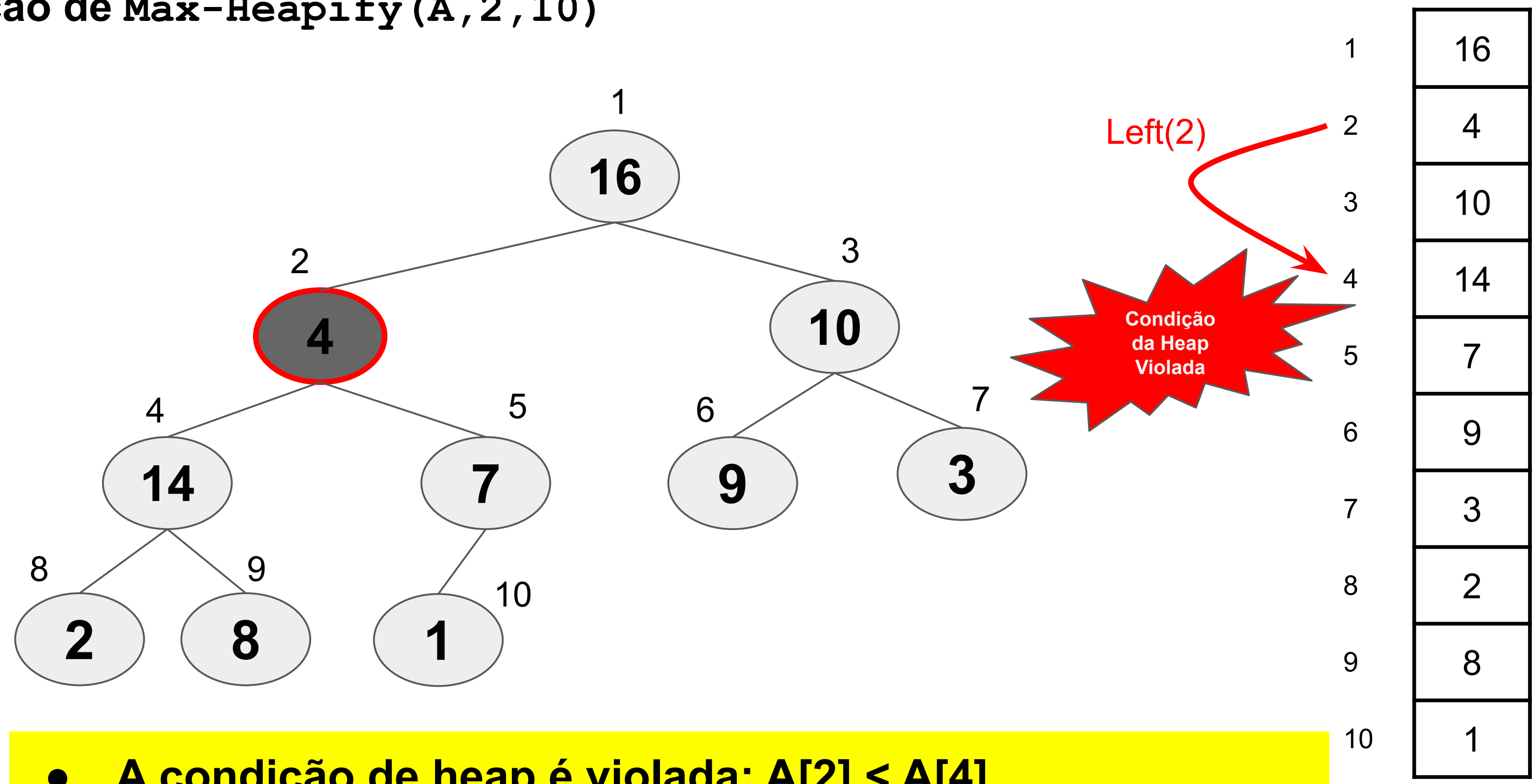
  A[Parent(i)] <= A[i]

  - O menor valor está na raiz.
- HeapSort usa heaps de máximos.

Manutenção da Propriedade de Heap Máximo

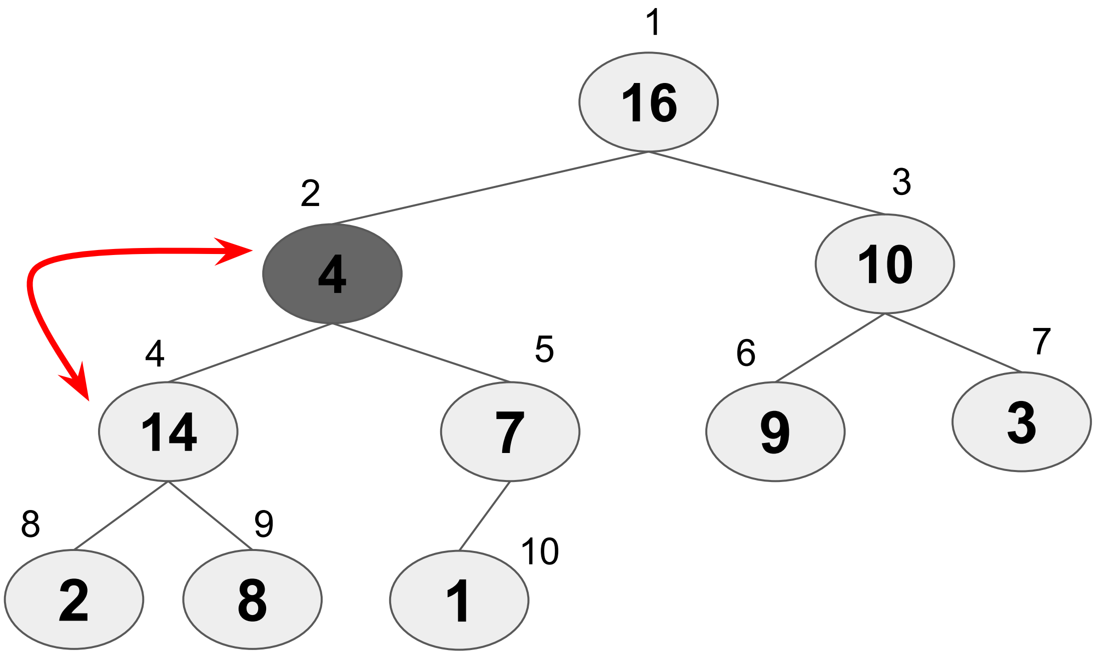
- Vamos definir o procedimento **`Max-Heapify`** que verifica se a propriedade de heap máximo não é violada em um determinado nó da árvore.
- Violação a ser verificada:
  - ***Dado um nó i, A[i] não pode ser menor que A[Left(i)] e A[Right(i)].***
- **`Max-Heapify`** permite que o valor de A[i] "*flutue para baixo*" no heap, de modo que a subárvore com raiz no índice i obedeça a propriedade do *heap* máximo.

Ação de `Max-Heapify(A,2,10)`



- A condição de heap é violada: A[2] < A[4]

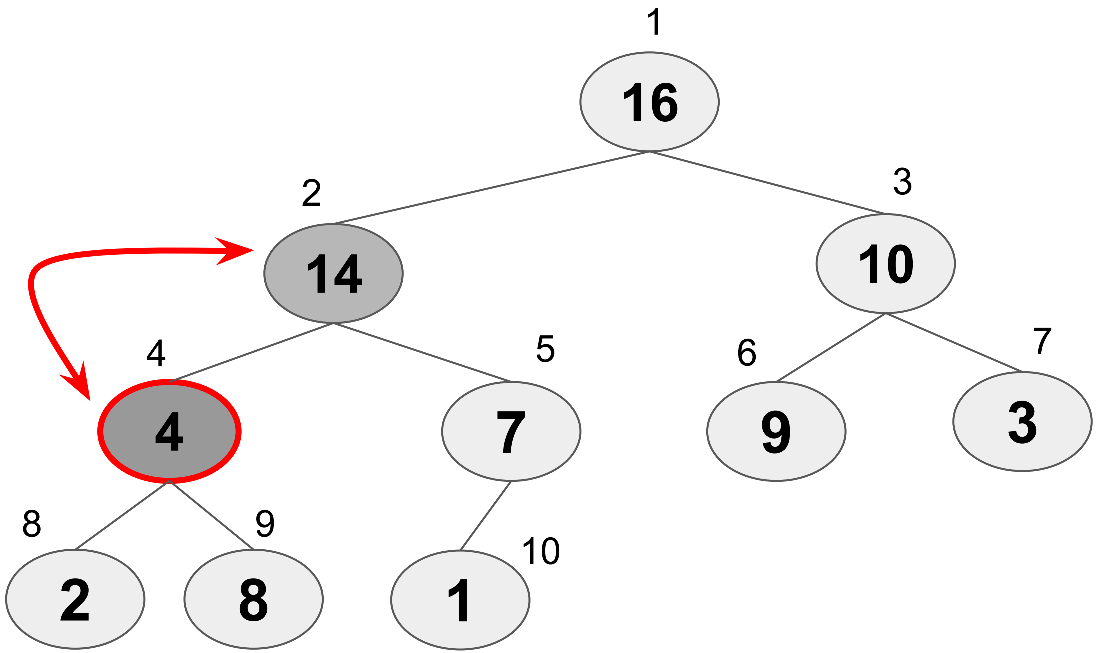
Ação de `Max-Heapify(A,2,10)`

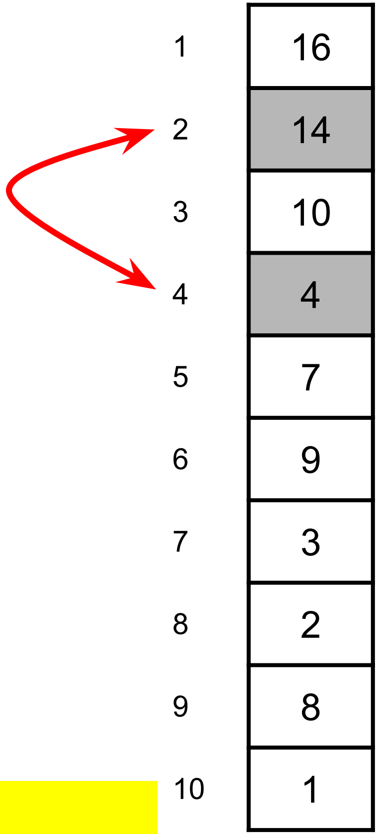


1	16
2	4
3	10
4	14
5	7
6	9
7	3
8	2
9	8
10	1

- A condição de heap é violada: A[2] < A[4]
    - **Troca A[2] com A[4];**

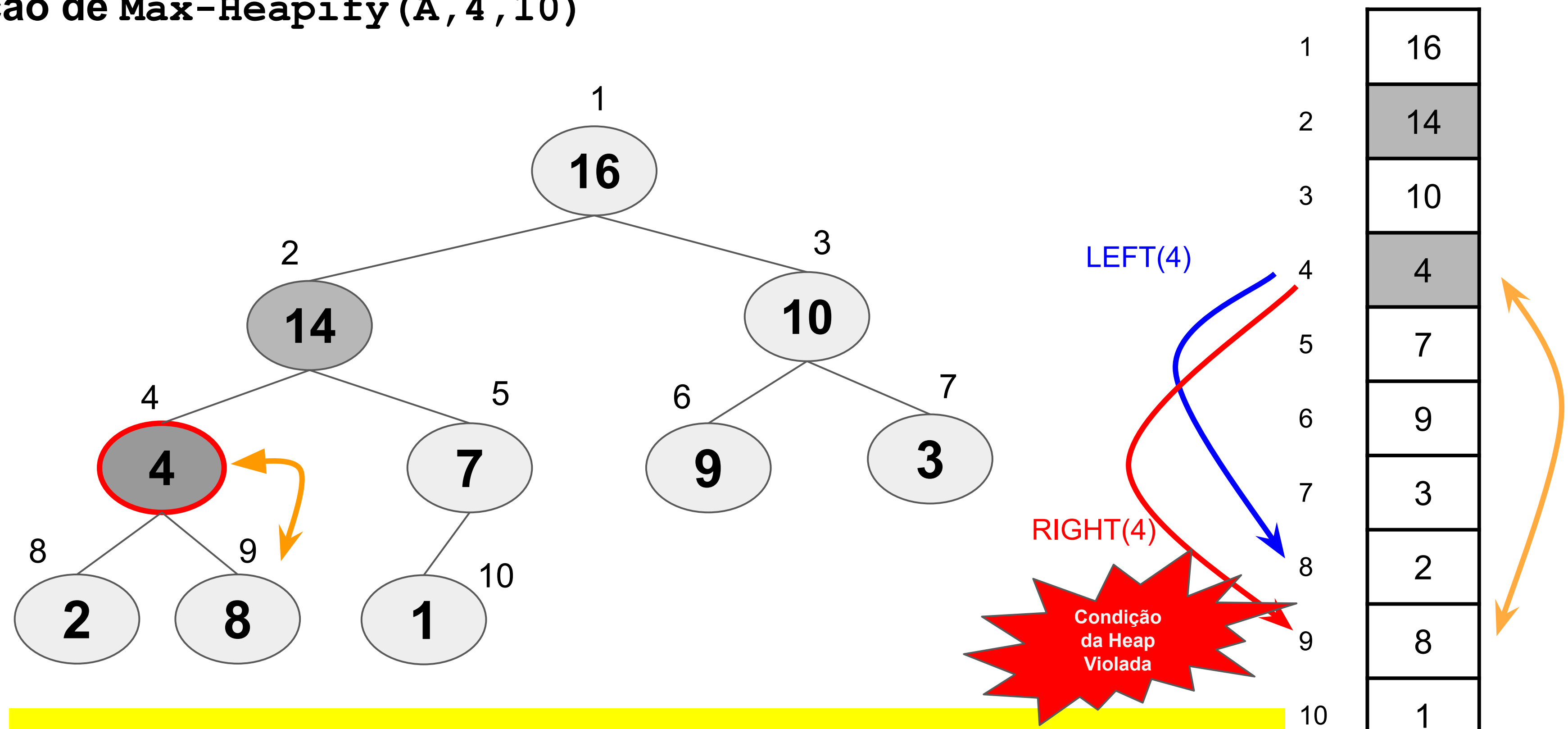
Ação de `Max-Heapify(A,2,10)`





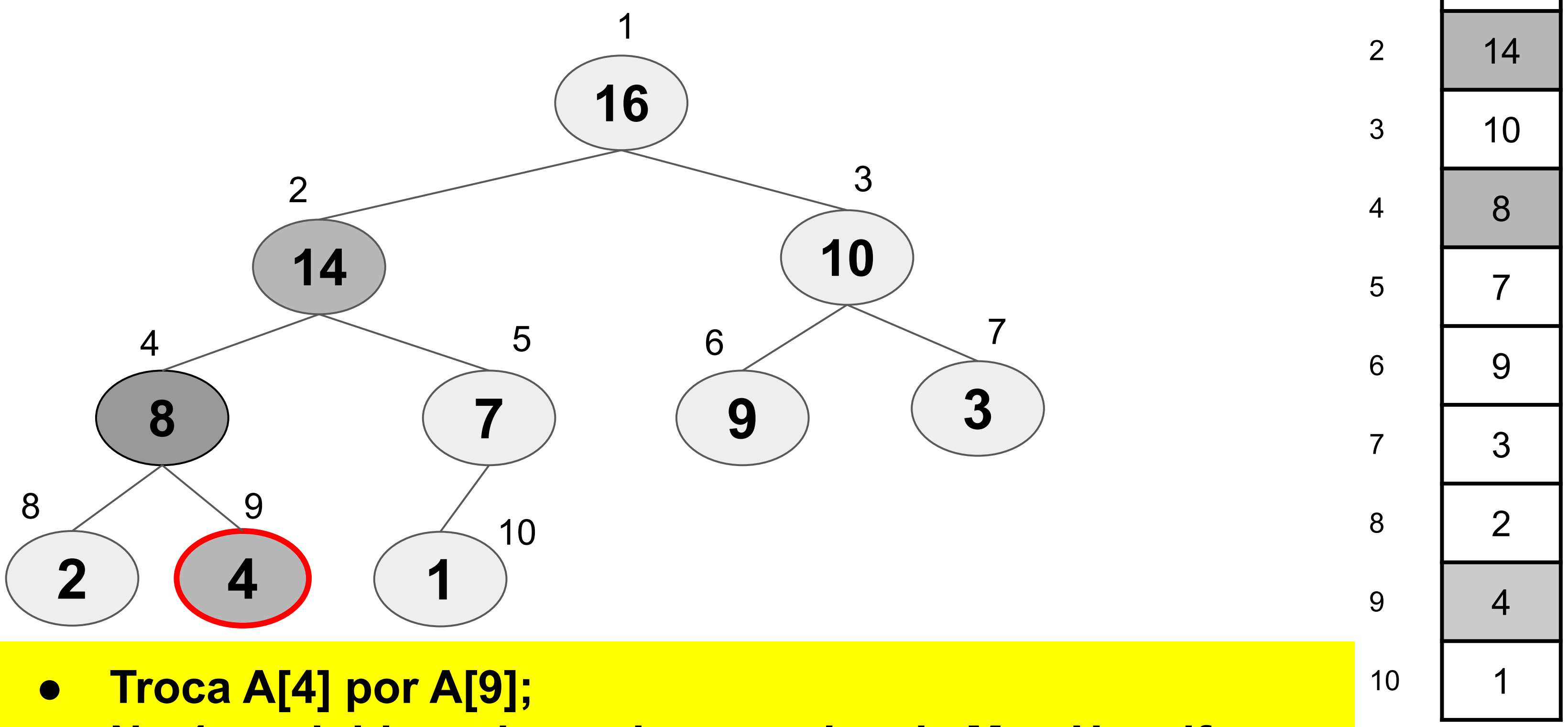
- **Verificar a ação do Max-Heapify na nó 4.**

Ação de `Max-Heapify(A,4,10)`



- Nó 2 é corrigido e devemos verificar a ação do Max-Heapify na nó 4.

Ação de `Max-Heapify(A,4,10)`



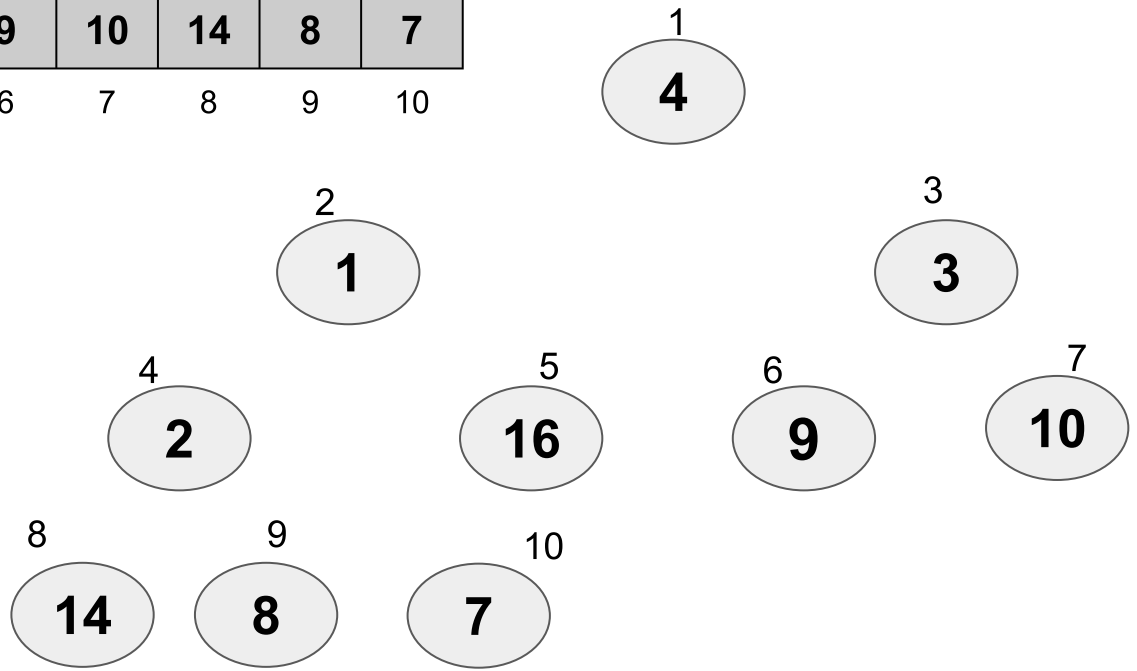
- **Troca A[4] por A[9];**
- **No 4 corrigido e chamada recursiva de Max-Heapify no nó 9 não produz nenhuma mudança na estrutura.**

Manutenção da Propriedade de Heap

```
Max-Heapify(A:vetor, i: int, n: int)
   l = Left(i)
   r = Right(i)
   Se ( (l <= n) E (A[l] > A[i]) ) Então m ← l
   Caso Contrário m ← i
   Fim-Se
   Se (r <= n E A[r] > A[m] ) Então m ← r
   Fim-Se
   Se (m != i ) Então
        trocar A[i] com A[m]
        Max-Heapify(A,m,n)
   Fim-Se
Fim
```

Construção da Heap

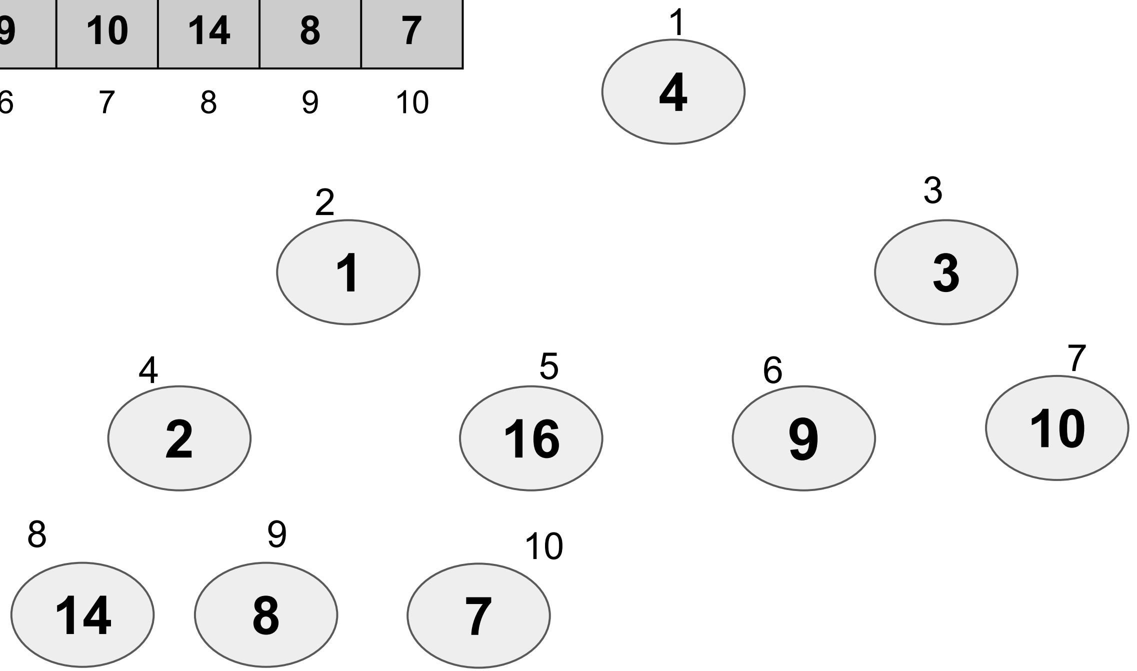
4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10



Note que os itens A[n/2 + 1], A[n/2 + 2], . . . , A[n] são folhas da árvore e, portanto, cada um deles é um heap de 1 elemento.

Construção da Heap

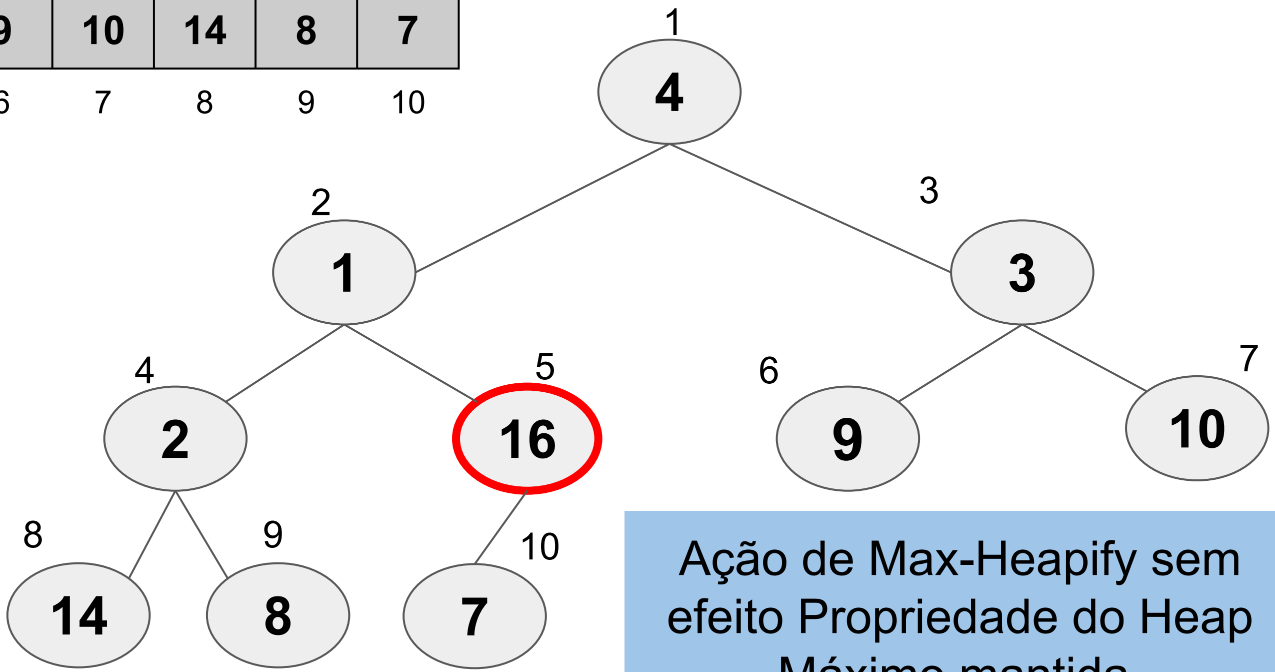
4	**1**	**3**	**2**	**16**	**9**	**10**	**14**	**8**	**7**
1	2	3	4	5	6	7	8	9	10



Em cada passo, os nós restantes A[n/2], A[n/2-1],.... A[1] são percorridos e Max-Heapify é executado sobre cada um deles

Construção da Heap

4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

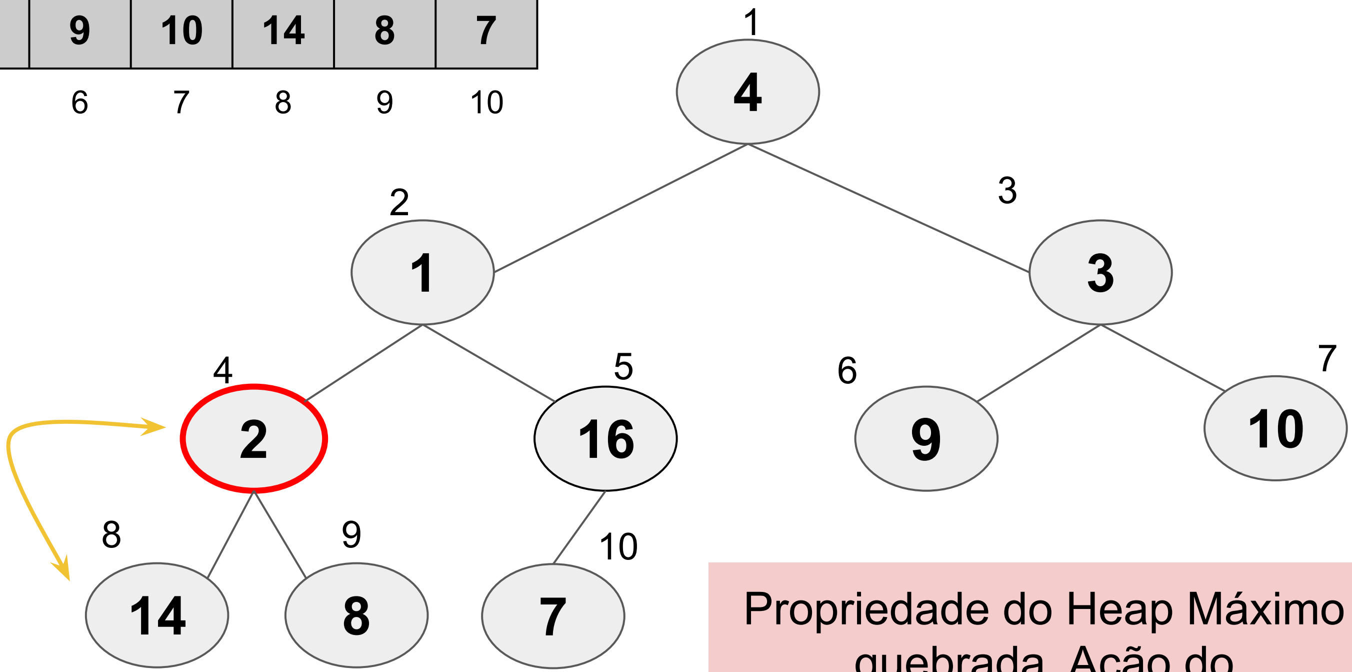


Ação de Max-Heapify sem efeito Propriedade do Heap Máximo mantida.

Incluído nó A[5] e chamando Max-Heapify(A,5,10)

Construção da Heap

4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

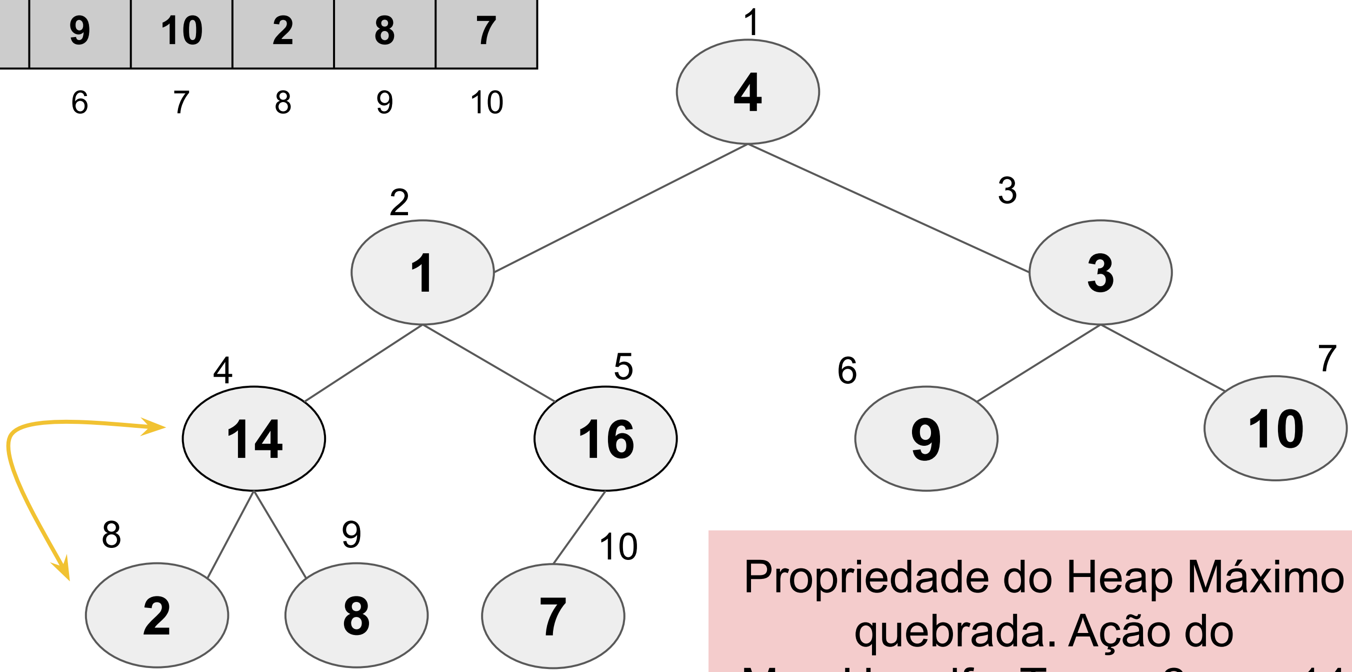


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 2 com 14

Incluído nó A[4] e chamando Max-Heapify(A,4,10)

Construção da Heap

4	1	3	14	16	9	10	2	8	7
1	2	3	4	5	6	7	8	9	10

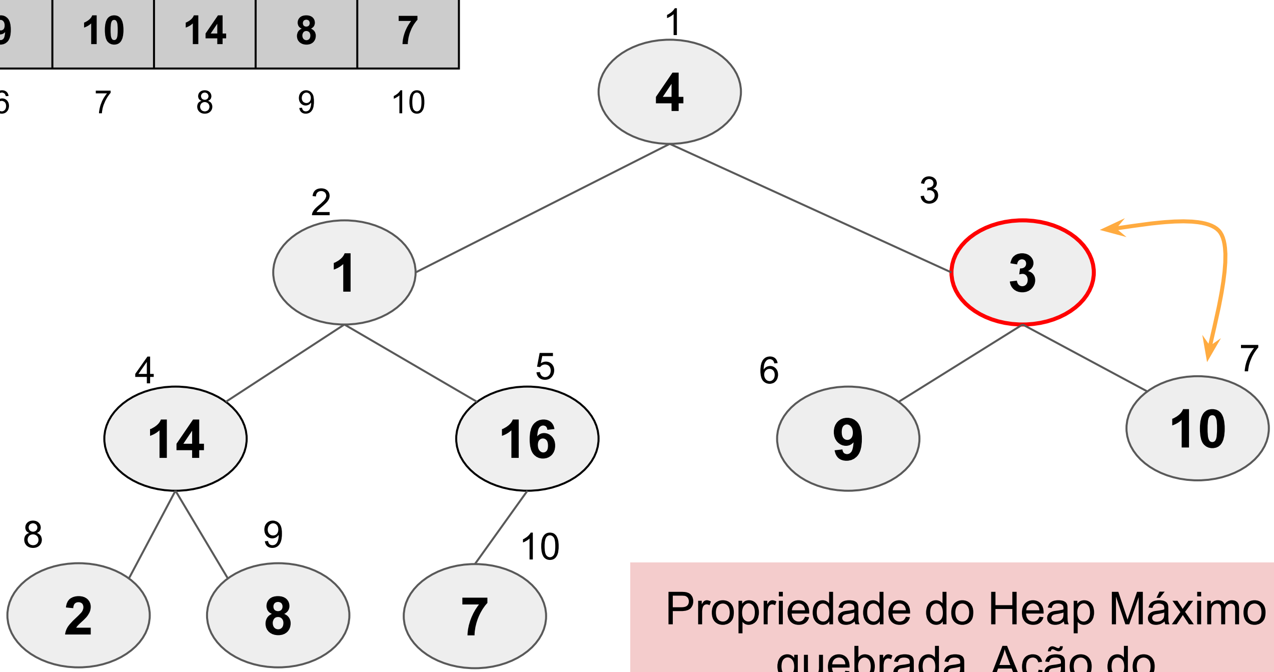


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 2 com 14

Incluído nó A[4] e chamando Max-Heapify(A,4,10)

Construção da Heap

4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

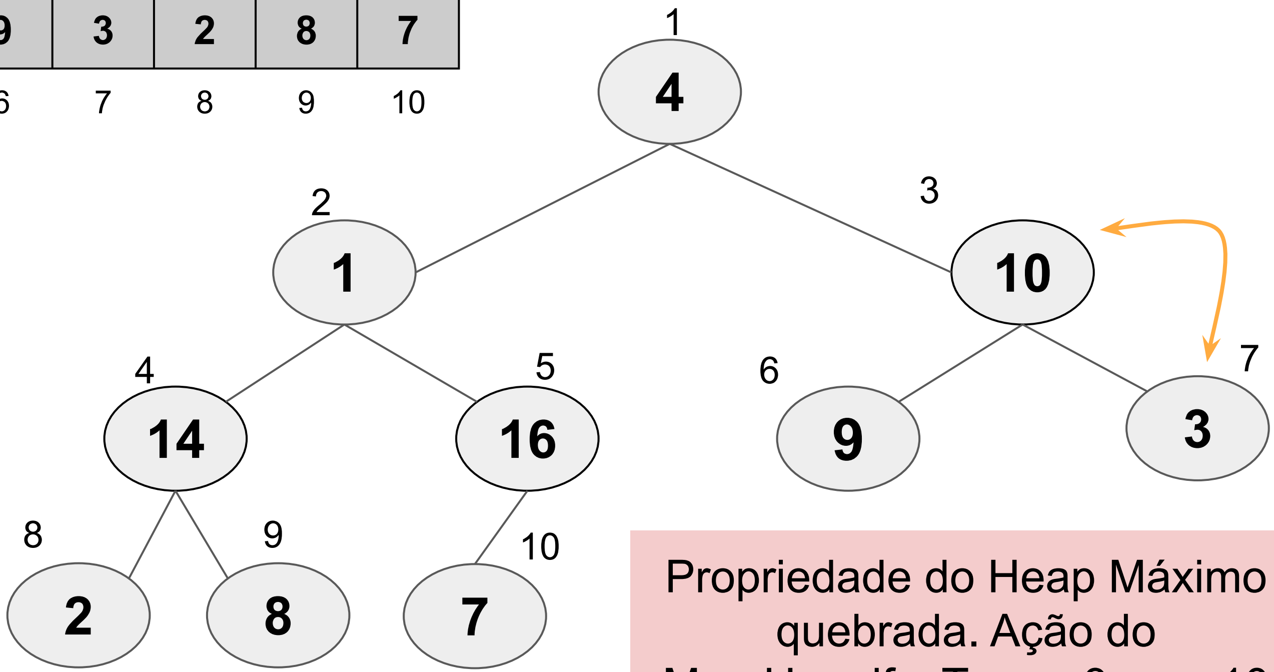


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 3 com 10

Incluído nó A[3] e chamando Max-Heapify(A,3,10)

Construção da Heap

4	1	10	14	16	9	3	2	8	7
1	2	3	4	5	6	7	8	9	10

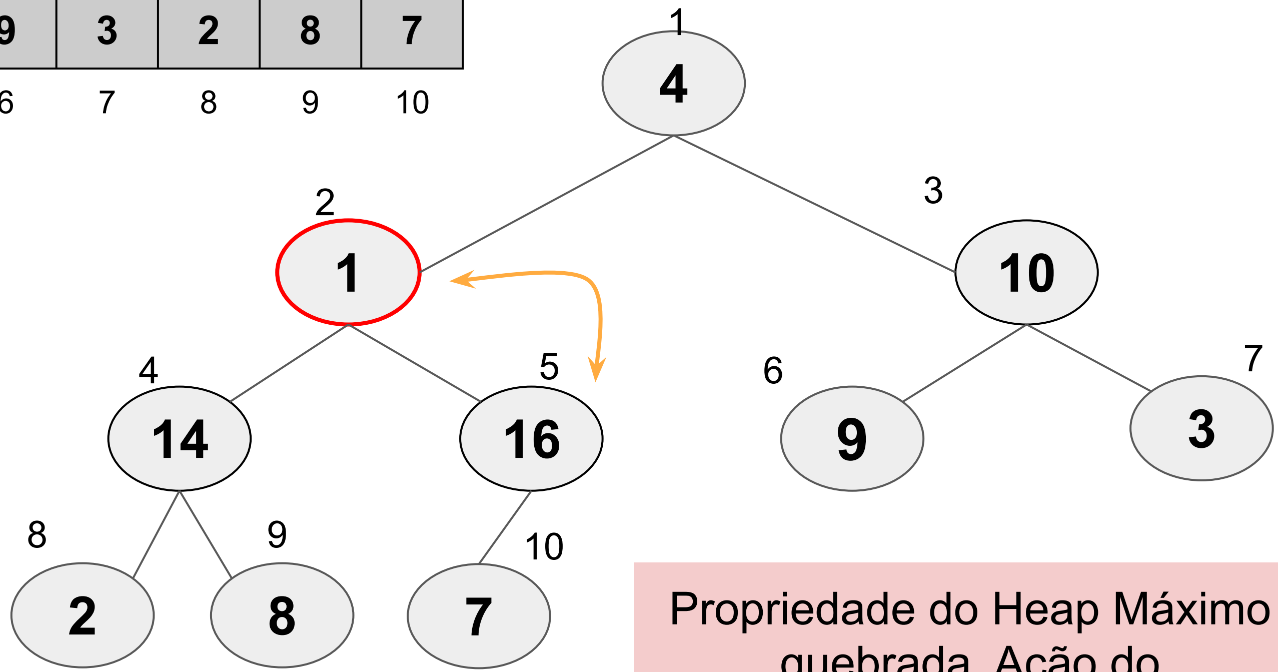


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 3 com 10

Incluído nó A[3] e chamando Max-Heapify(A,3,10)

Construção da Heap

4	**1**	**10**	**14**	**16**	**9**	**3**	**2**	**8**	**7**
1	2	3	4	5	6	7	8	9	10

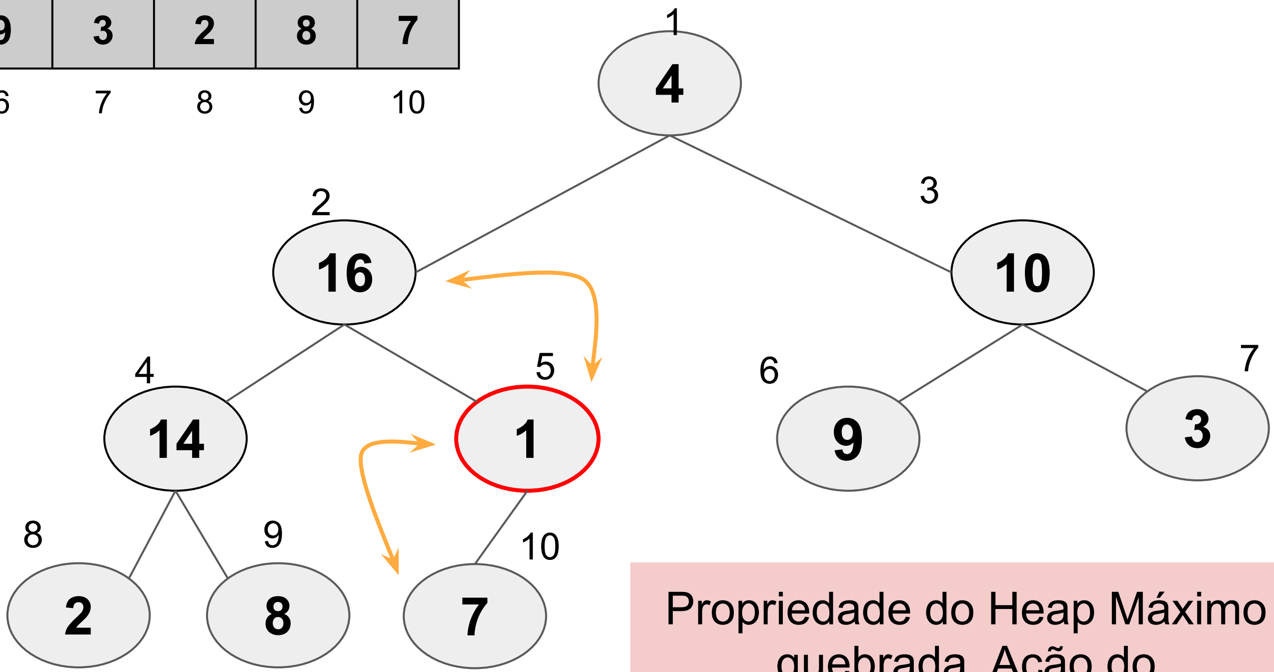


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 1 com 16

Incluído nó A[2] e chamando Max-Heapify(A,2,10)

Construção da Heap

4	1	10	14	16	9	3	2	8	7
1	2	3	4	5	6	7	8	9	10

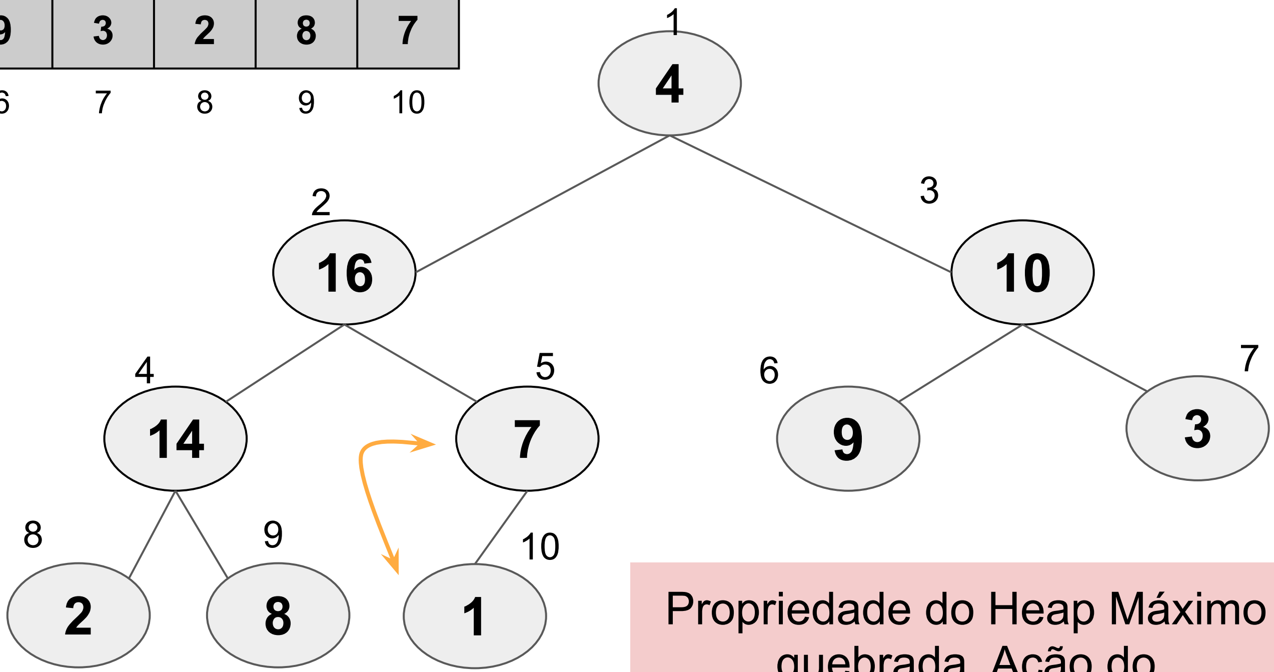


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 1 com 7

Incluído nó A[2] e chamando Max-Heapify(A,2,10)

Construção da Heap

4	16	10	14	1	9	3	2	8	7
1	2	3	4	5	6	7	8	9	10

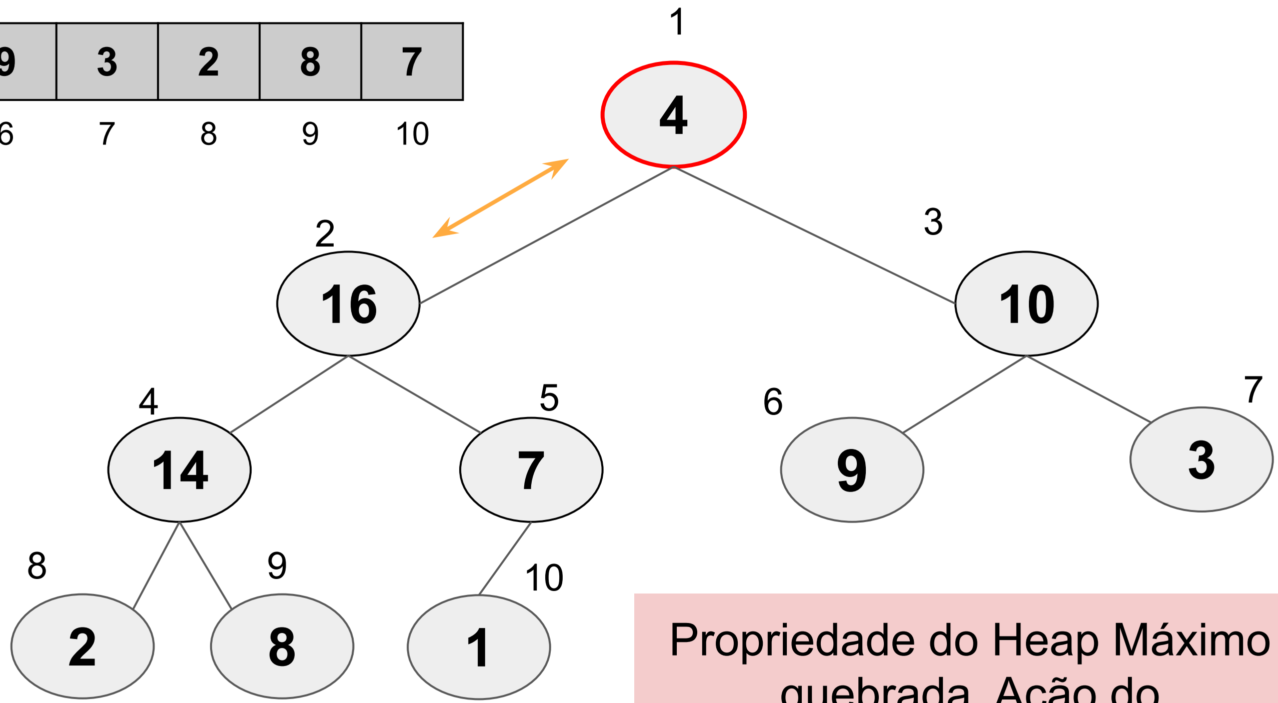


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 1 com 7

Incluído nó A[2] e chamando Max-Heapify(A,2,10)

Construção da Heap

4	16	10	14	1	9	3	2	8	7
1	2	3	4	5	6	7	8	9	10

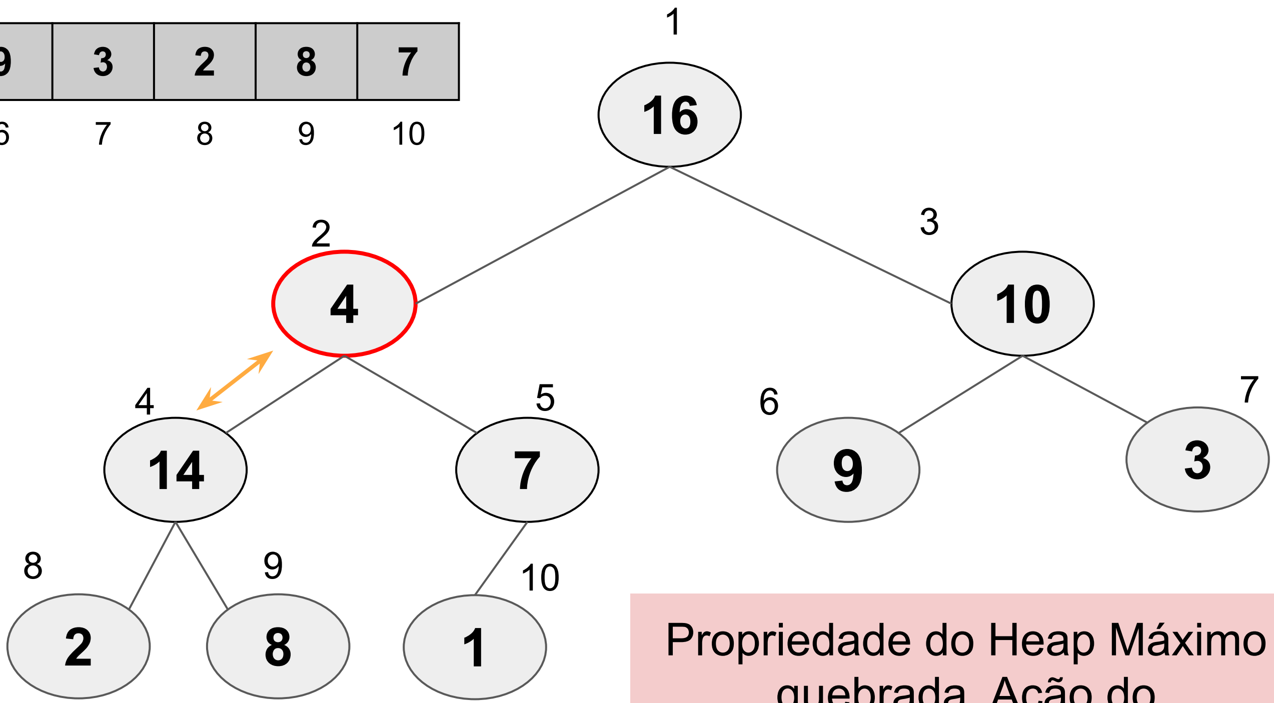


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 4 com 16

Incluído nó A[1] e chamando Max-Heapify(A,1,10)

Construção da Heap

16	4	10	14	1	9	3	2	8	7
1	2	3	4	5	6	7	8	9	10

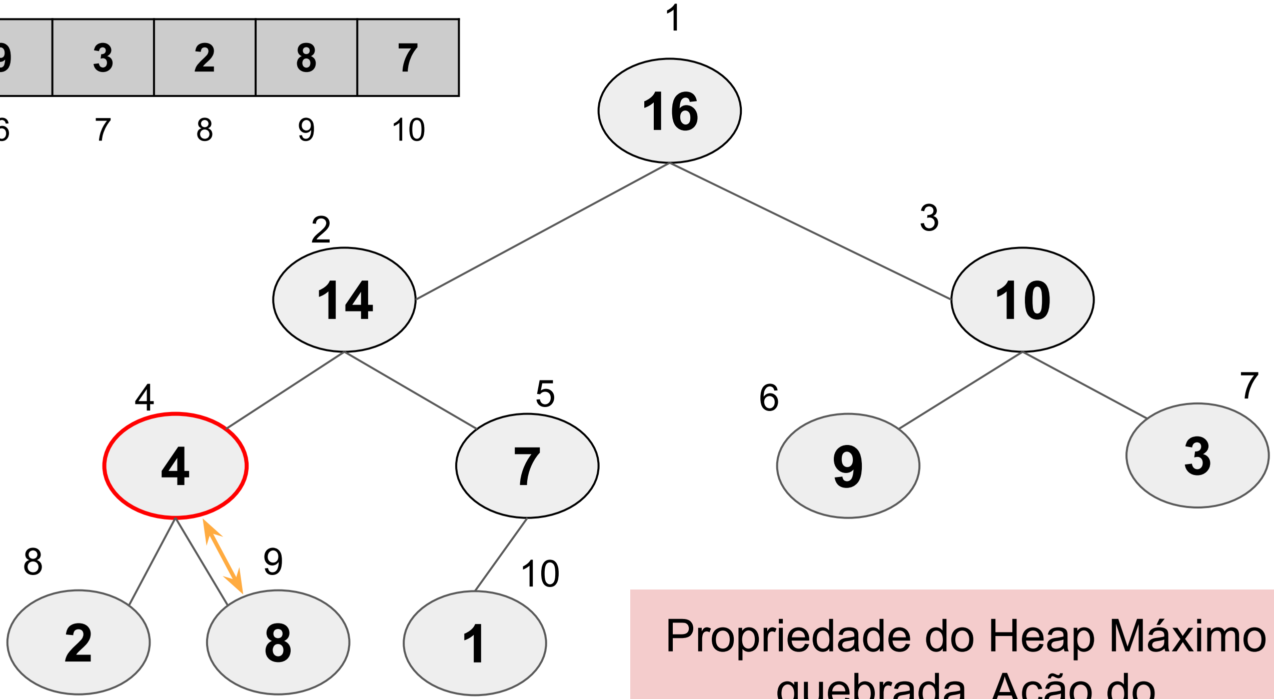


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 4 com 14

Incluído nó A[1] e chamando Max-Heapify(A,1,10)

Construção da Heap

16	**14**	**10**	**4**	**1**	**9**	**3**	**2**	**8**	**7**
1	2	3	4	5	6	7	8	9	10

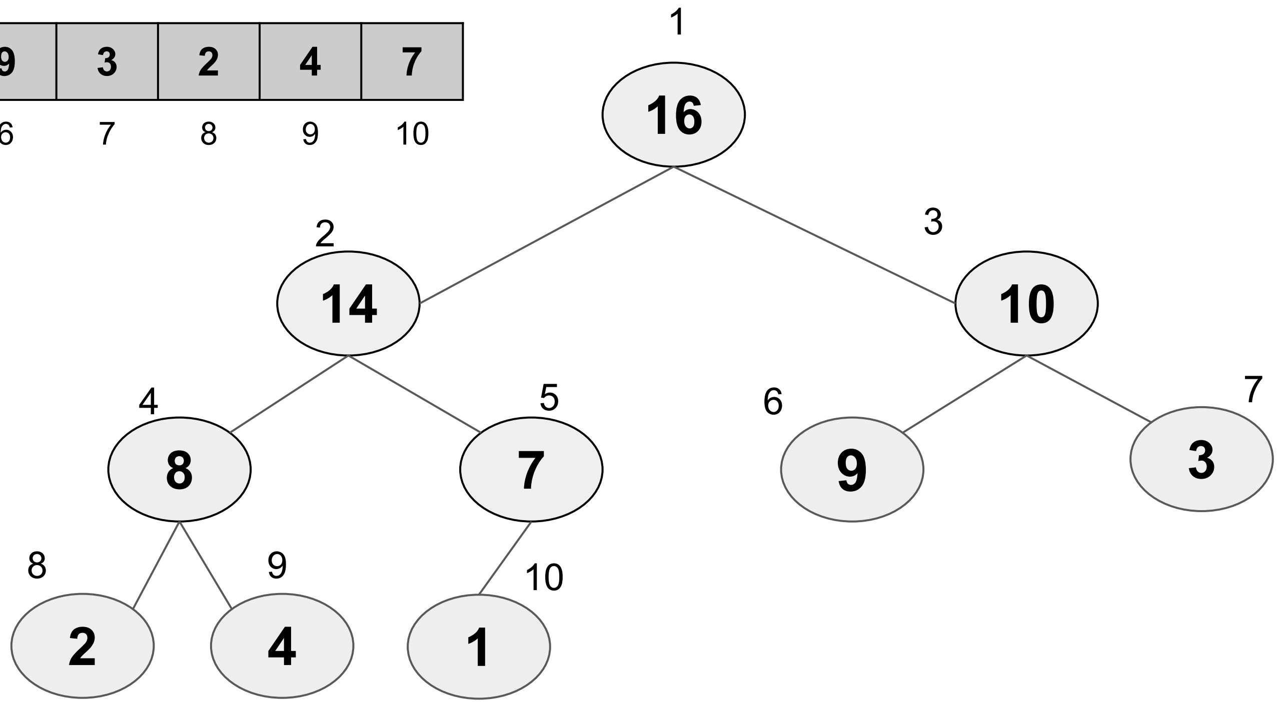


Propriedade do Heap Máximo quebrada. Ação do Max-Heapify: Trocar 4 com 8

Incluído nó A[1] e chamando Max-Heapify(A,1,10)

Construção da Heap

16	14	10	8	1	9	3	2	4	7
1	2	3	4	5	6	7	8	9	10



Fim da Construção da Heap

Construção de um Heap

- Usa-se o procedimento **`Max-Heapify`** para converter um arranjo A[1,...,n] em um heap máximo.
- Note que os itens A[n/2 + 1], A[n/2 + 2], . . . , A[n] formam um heap:
  - Neste intervalo não existem dois índices l e r tais que $l = 2i$ ou $r = 2i + 1$.
  - Eles são folhas da árvore e, portanto, cada um deles é um heap de 1 elemento.
- Assim, em cada passo, os nós restantes A[n/2], A[n/2-1],.... A[1] são percorridos e **`Max-Heapify`** é executado sobre cada um.

Construção de um Heap

```
Build-Max-Heap(A:vetor,n: int)
    Para i = n/2 até 1 faça
        Max-Heapify(A,i,n)
    Fim-Para
Fim
```

HeapSort

➢ Algoritmo:

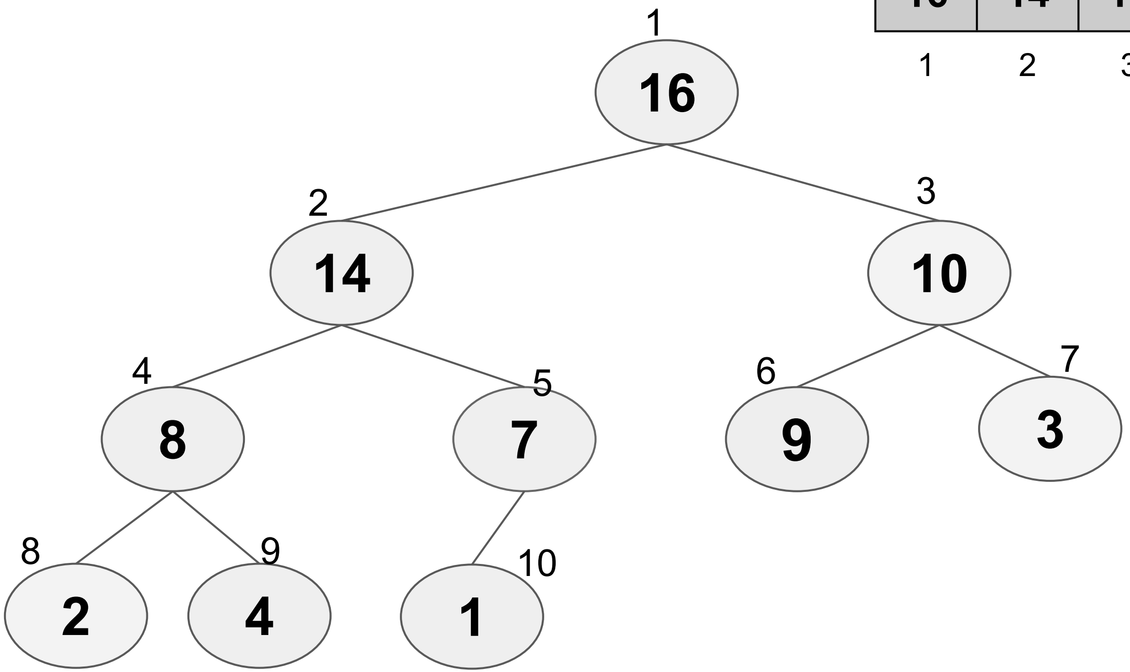
1. Construir a `Heap: Max-build-Heap`
2. Troque o item na posição 1 do vetor (raiz do heap) com o item da posição n.
3. Use o procedimento **`Max-Heapify`** para reconstituir o heap para os itens A[1], A[2], . . . , A[n − 1].
4. Repita os passos 2 e 3 com os n − 1 itens restantes, depois com os n − 2, até que reste apenas um item.

HeapSort: Algoritmo

```
HeapSort(A:vetor,n: int)
    Build-Max-Heap(A)
    Para i = n até 2 Faça
        Trocar A[1] com A[i]
        Max-Heapify(A,1,n-1)
    Fim-Para
Fim
```

Operação do HeapSort

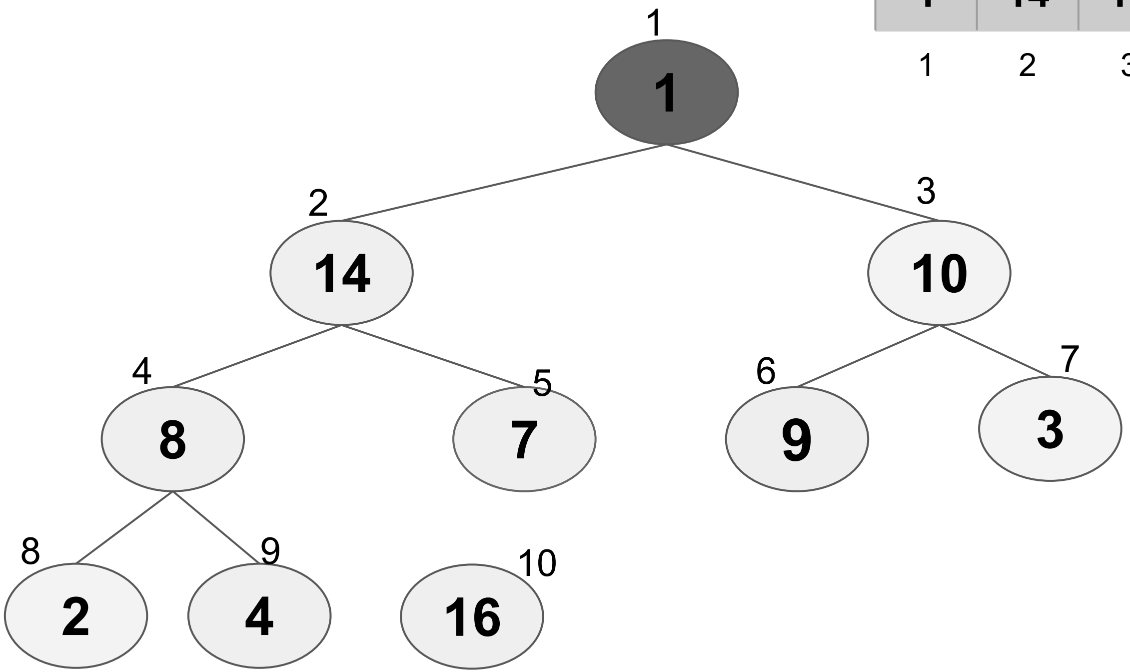
16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10



- Após a chamada Build-Max-Heap
- Laço aponta para o nó 10. Troca A[1] com A[10].

Operação do HeapSort

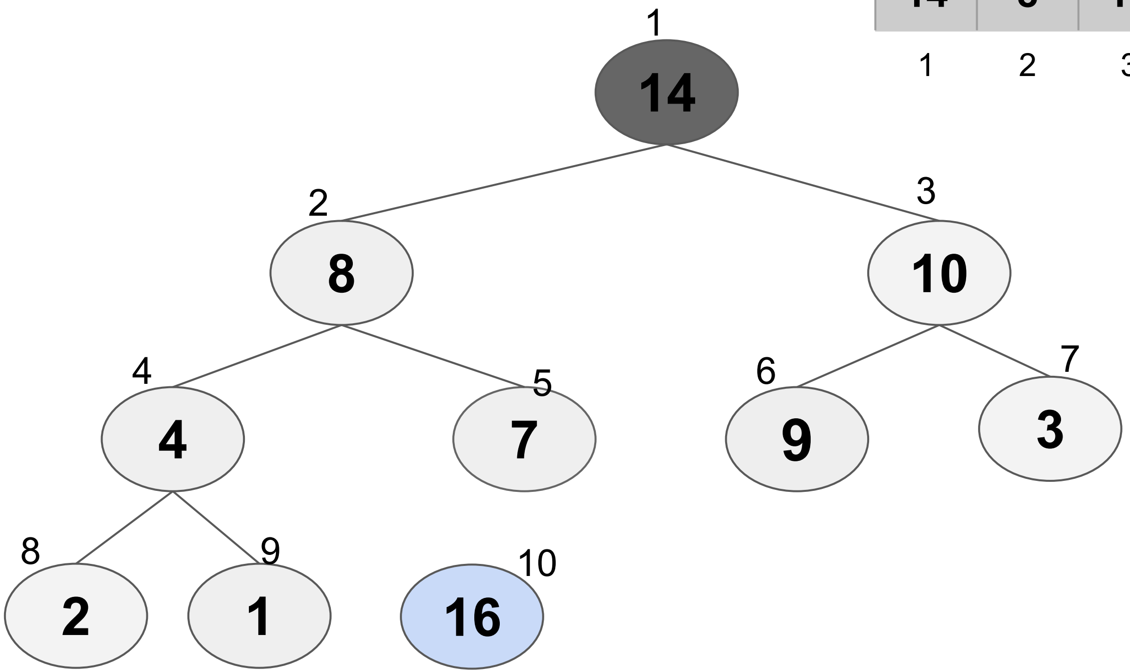
1	14	10	8	7	9	3	2	4	16
1	2	3	4	5	6	7	8	9	10



- Após a chamada Build-Max-Heap
- Laço aponta para o nó 10. Troca A[1] com A[10].
- Chama Max-Heapify(A,1,9)

Operação do HeapSort

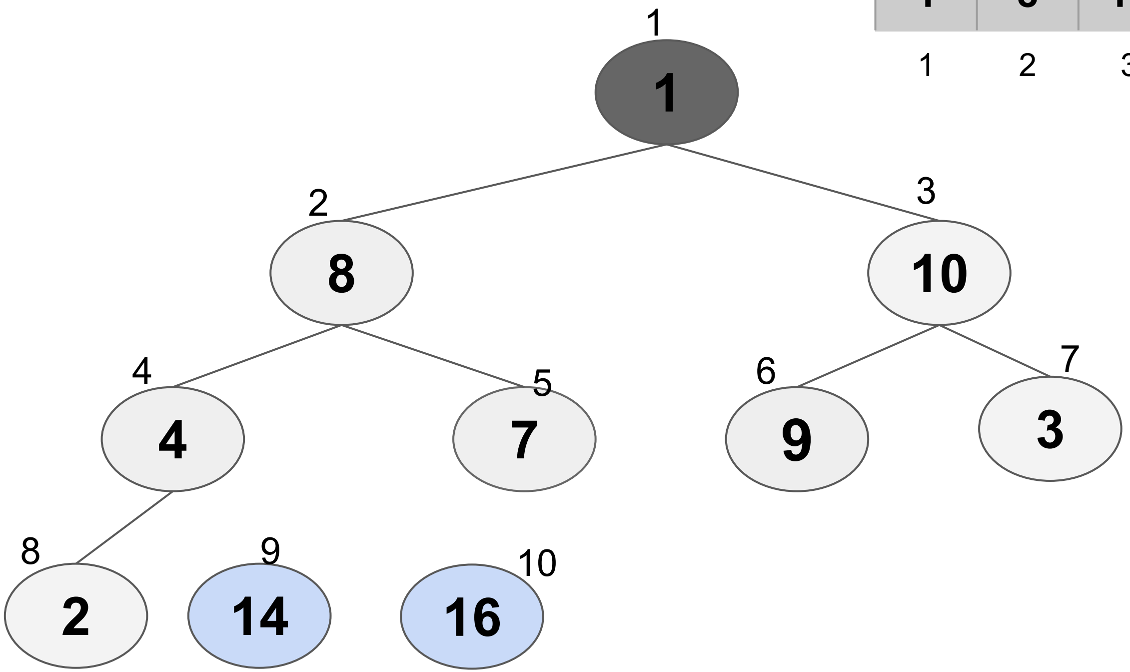
14	**8**	**10**	**4**	**7**	**9**	**3**	**2**	**1**	**16**
1	2	3	4	5	6	7	8	9	10



- **Após a chamada Max-Heapify(A,1,9)**

Operação do HeapSort

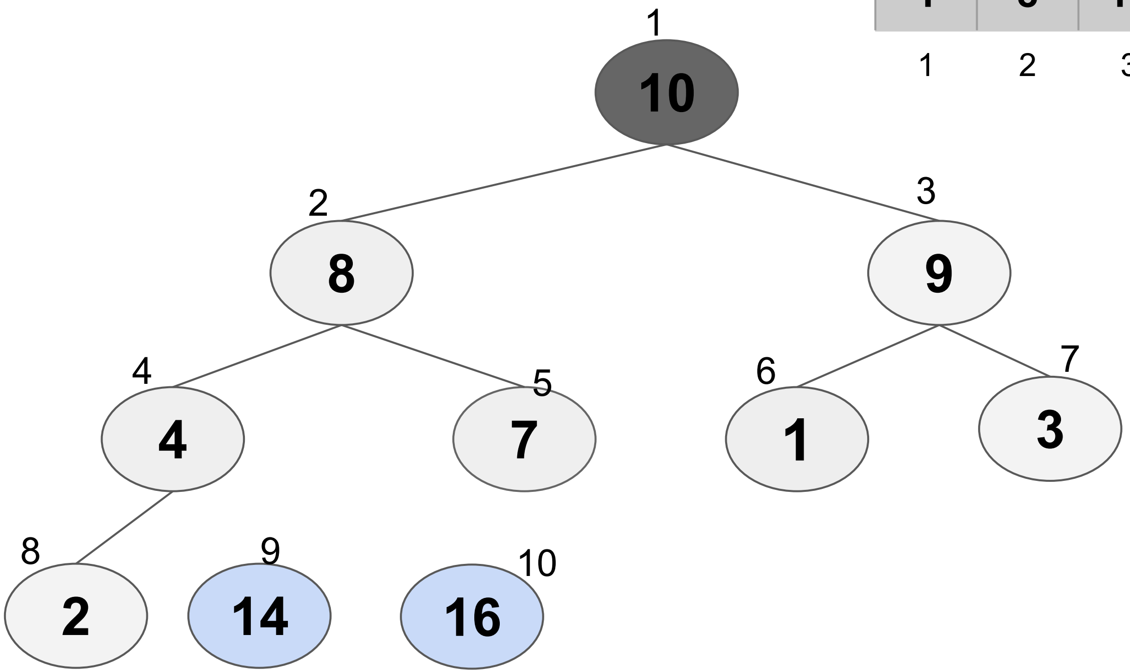
1	**8**	**10**	**4**	**7**	**9**	**3**	**2**	**14**	**16**
1	2	3	4	5	6	7	8	9	10



- **Após a chamada Max-Heapify(A,1,9)**
- **Laço aponta para o nó 9. Troca A[1] com A[9].**
- **Chama Max-Heapify(A,1,8)**

Operação do HeapSort

1	8	10	4	7	9	3	2	14	10
1	2	3	4	5	6	7	8	9	10

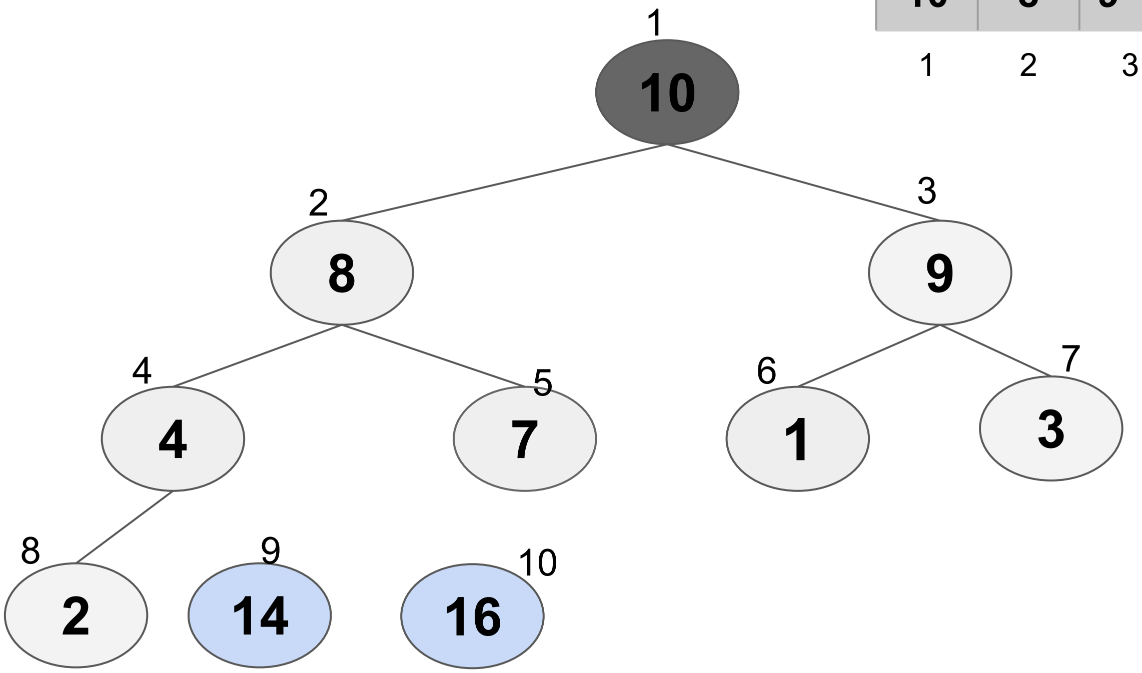


- **Após a chamada Max-Heapify(A,1,8)**

Operação do HeapSort

10	**8**	**9**	**4**	**7**	**1**	**3**	**2**	**14**	**10**
1	2	3	4	5	6	7	8	9	10

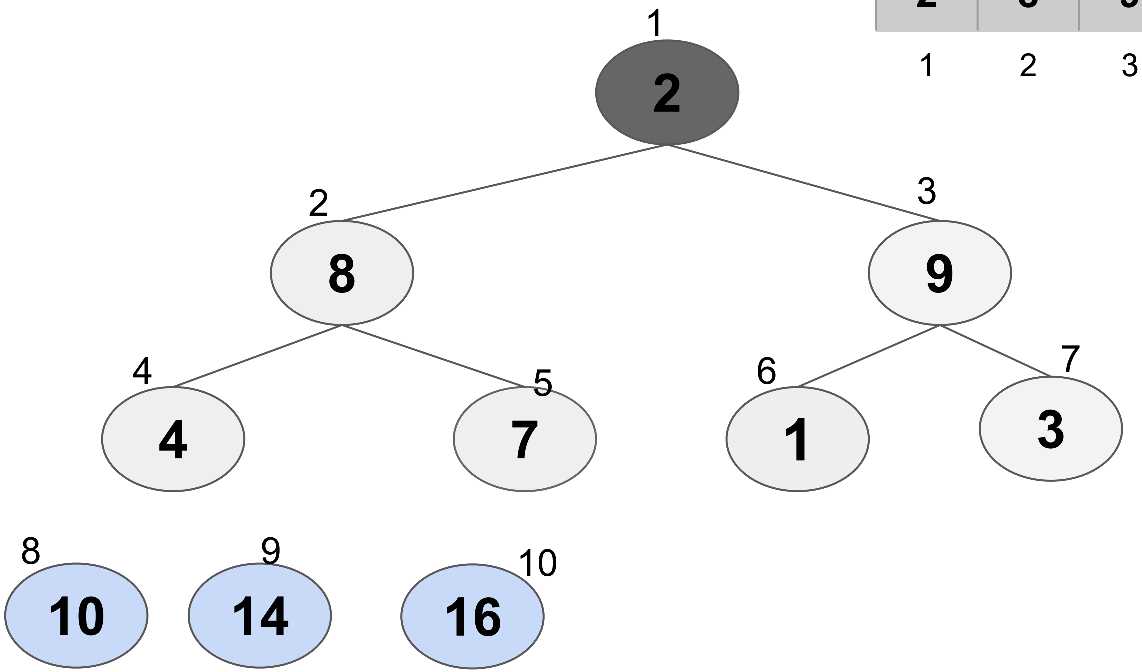
i



- **Após a chamada Max-Heapify(A,1,8)**

Operação do HeapSort

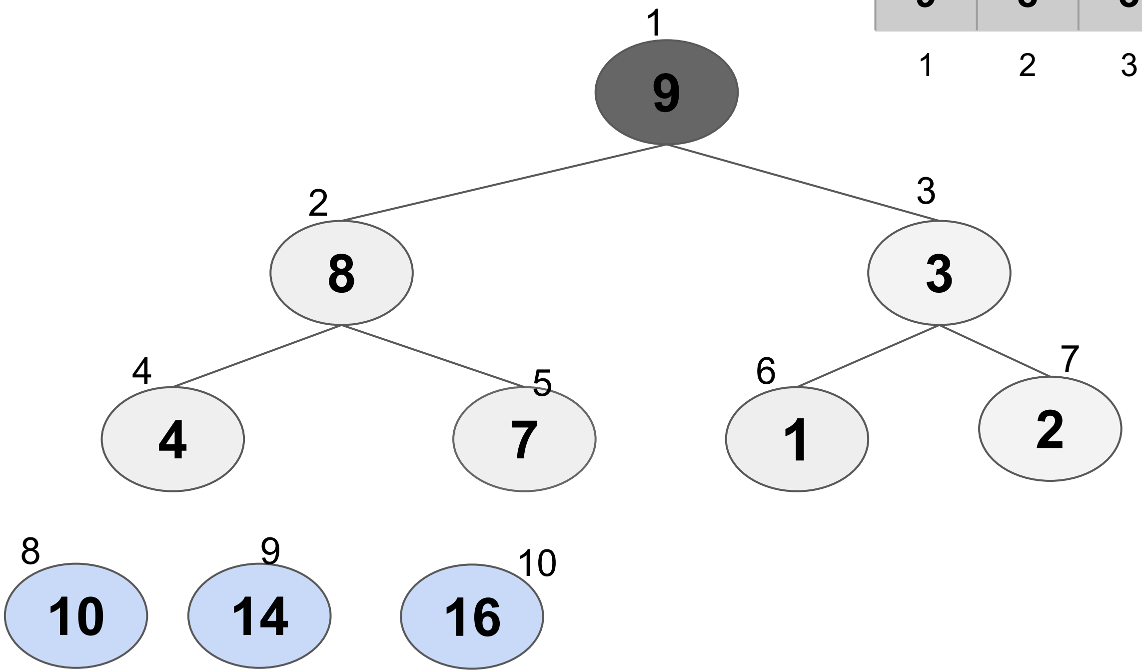
2	8	9	4	7	1	3	10	14	16
1	2	3	4	5	6	7	8	9	10



- **Após a chamada Max-Heapify(A,1,8)**
- **Laço aponta para o nó 8. Troca A[1] com A[8].**
- **Chama Max-Heapify(A,1,7)**

Operação do HeapSort

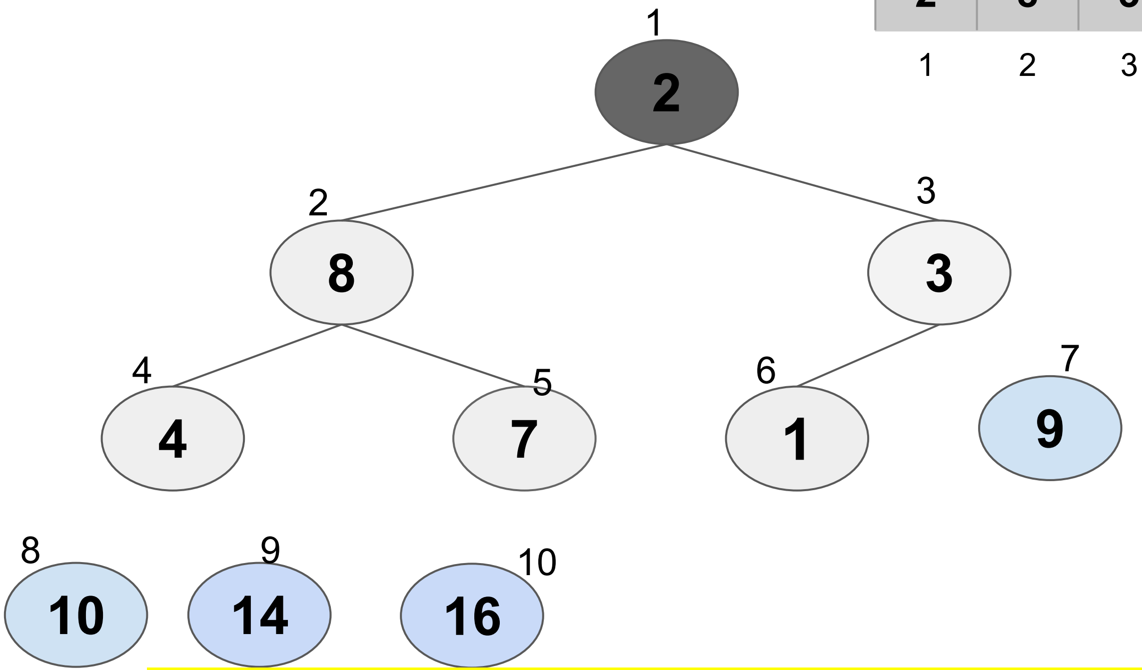
9	8	3	4	7	1	2	10	14	16
1	2	3	4	5	6	7	8	9	10



- **Após a chamada Max-Heapify(A,1,7)**

Operação do HeapSort

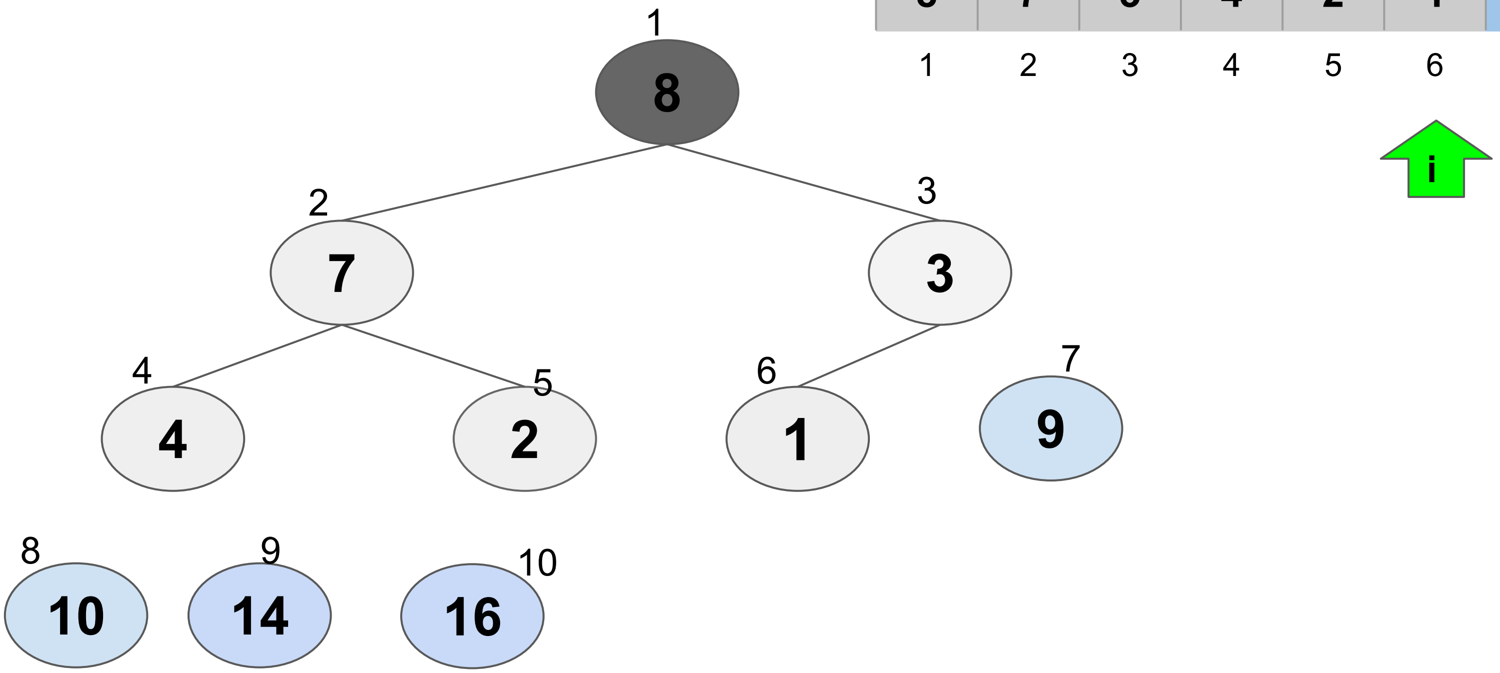
2	8	3	4	7	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10



- Após a chamada Max-Heapify(A,1,7)
- Laço aponta para o nó 7. Troca A[1] com A[7].
- Chama Max-Heapify(A,1,6)

Operação do HeapSort

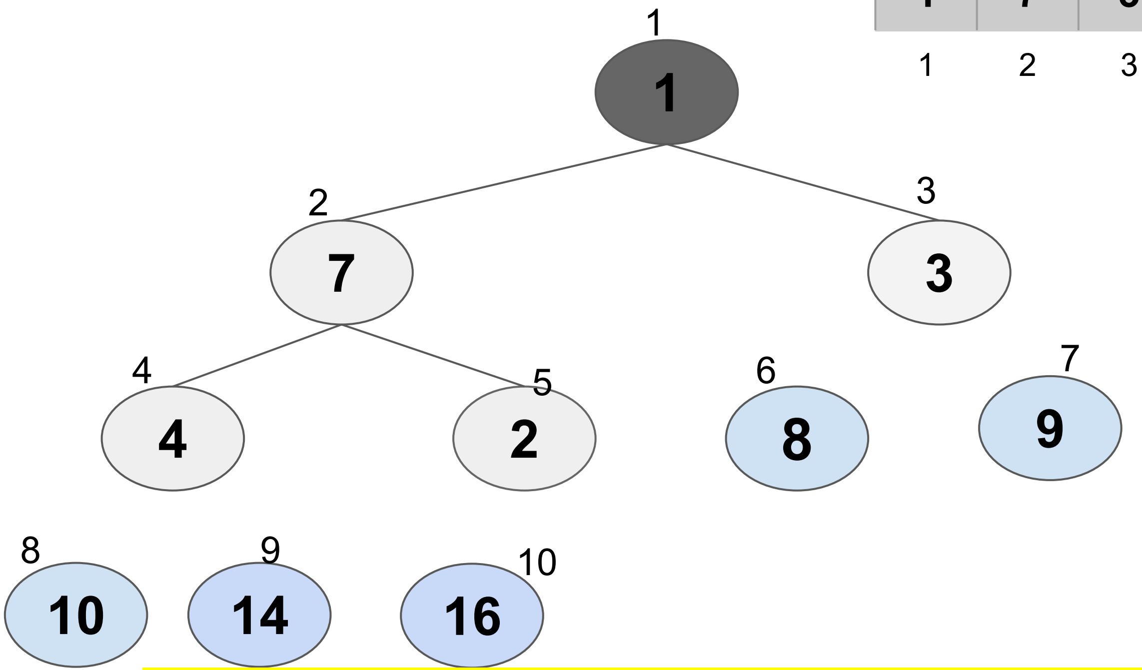
8	7	3	4	2	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10



- **Após a chamada Max-Heapify(A,1,6)**

Operação do HeapSort

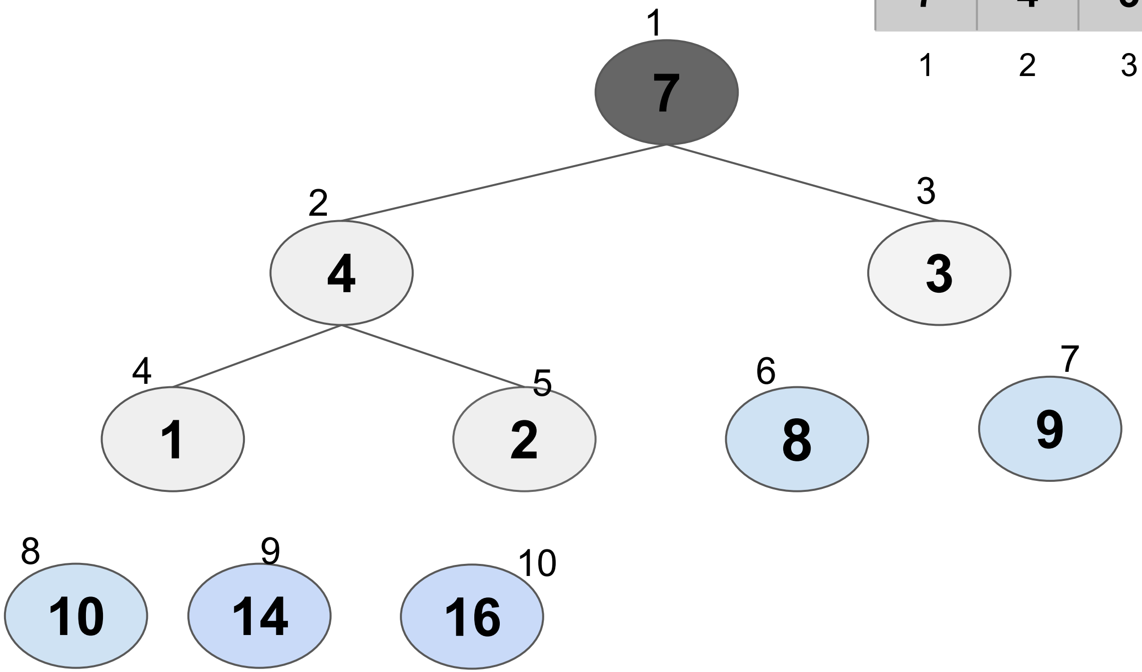
1	7	3	4	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10



- Após a chamada Max-Heapify(A,1,6)
- Laço aponta para o nó 6. Troca A[1] com A[6].
- Chama Max-Heapify(A,1,5)

Operação do HeapSort

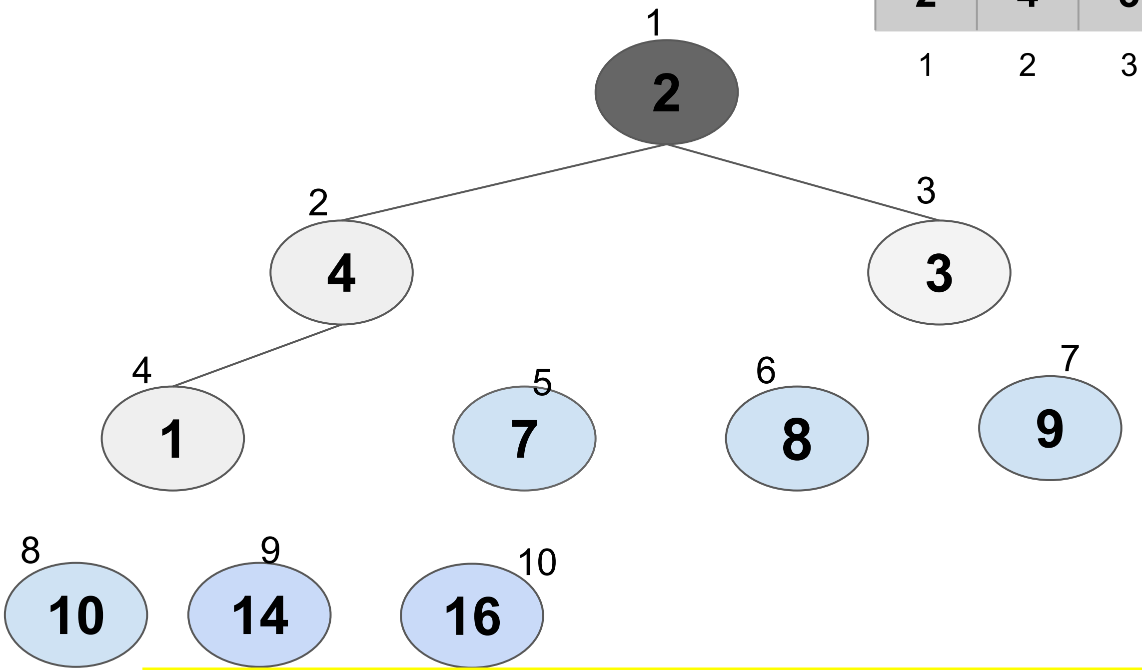
7	4	3	1	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10



- **Após a chamada Max-Heapify(A,1,5)**

Operação do HeapSort

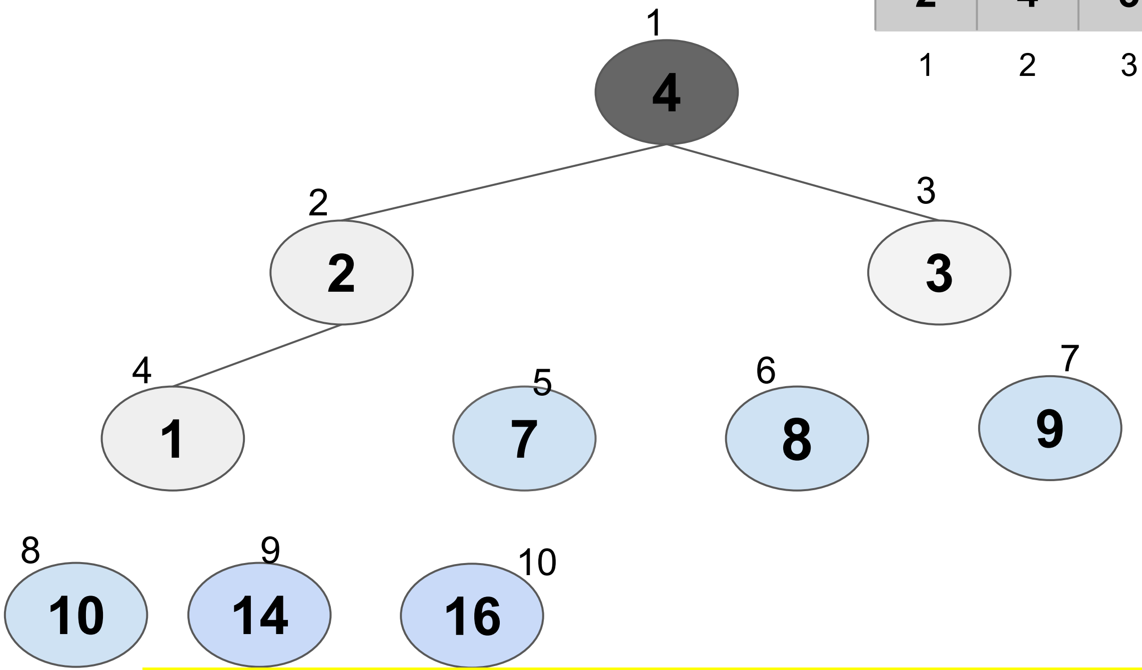
2	4	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10



- **Após a chamada Max-Heapify(A,1,5)**
- **Laço aponta para o nó 5. Troca A[1] com A[5].**
- **Chama Max-Heapify(A,1,4)**

Operação do HeapSort

2	**4**	**3**	**1**	**7**	**8**	**9**	**10**	**14**	**16**
1	2	3	4	5	6	7	8	9	10

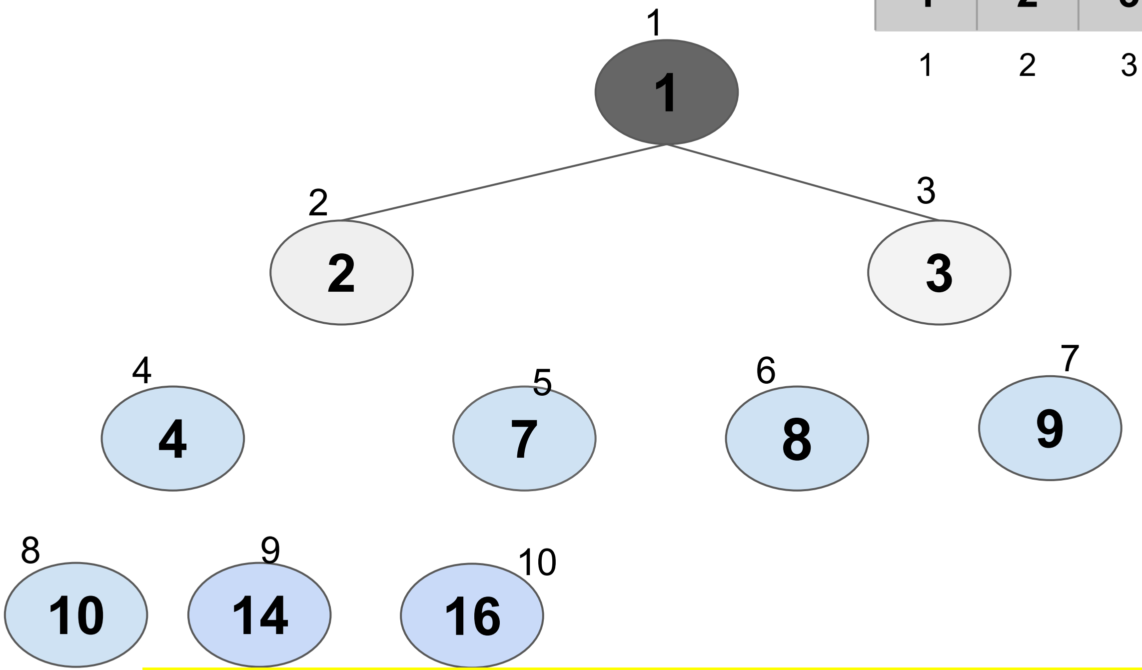


- Após a chamada Max-Heapify(A,1,5)
- Laço aponta para o nó 5. Troca A[1] com A[5].
- Chama Max-Heapify(A,1,4)

Operação do HeapSort

1	**2**	**3**	**4**	**7**	**8**	**9**	**10**	**14**	**16**
1	2	3	4	5	6	7	8	9	10

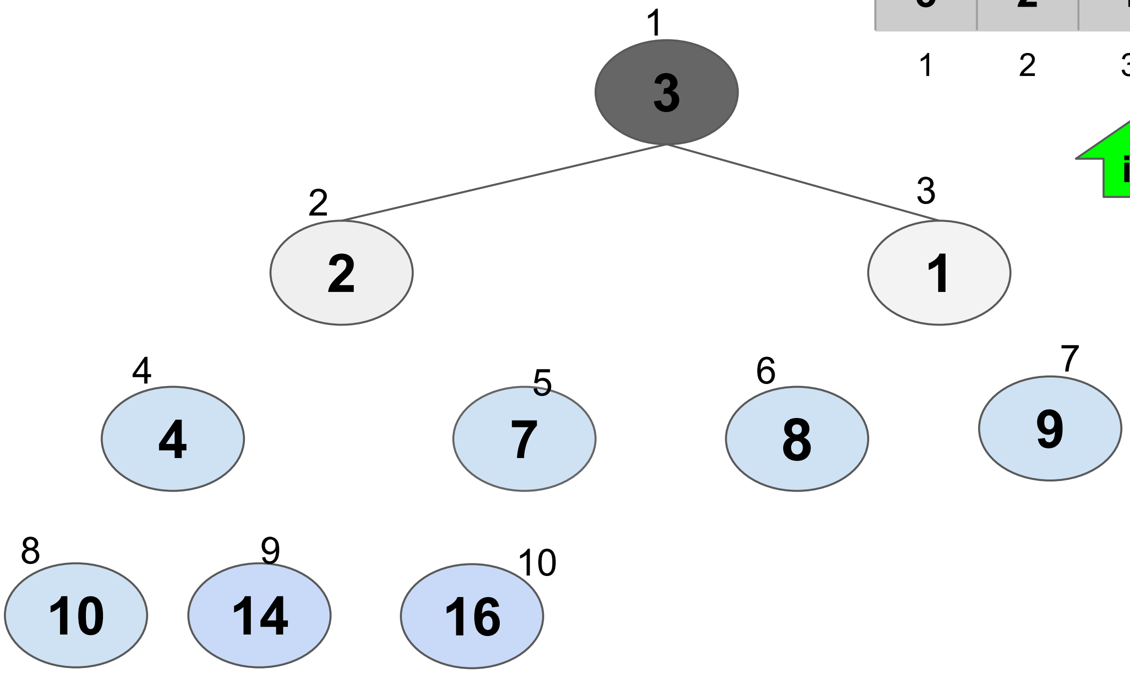
i



- **Após a chamada Max-Heapify(A,1,4)**
- **Laço aponta para o nó 4. Troca A[1] com A[4].**
- **Chama Max-Heapify(A,1,3)**

Operação do HeapSort

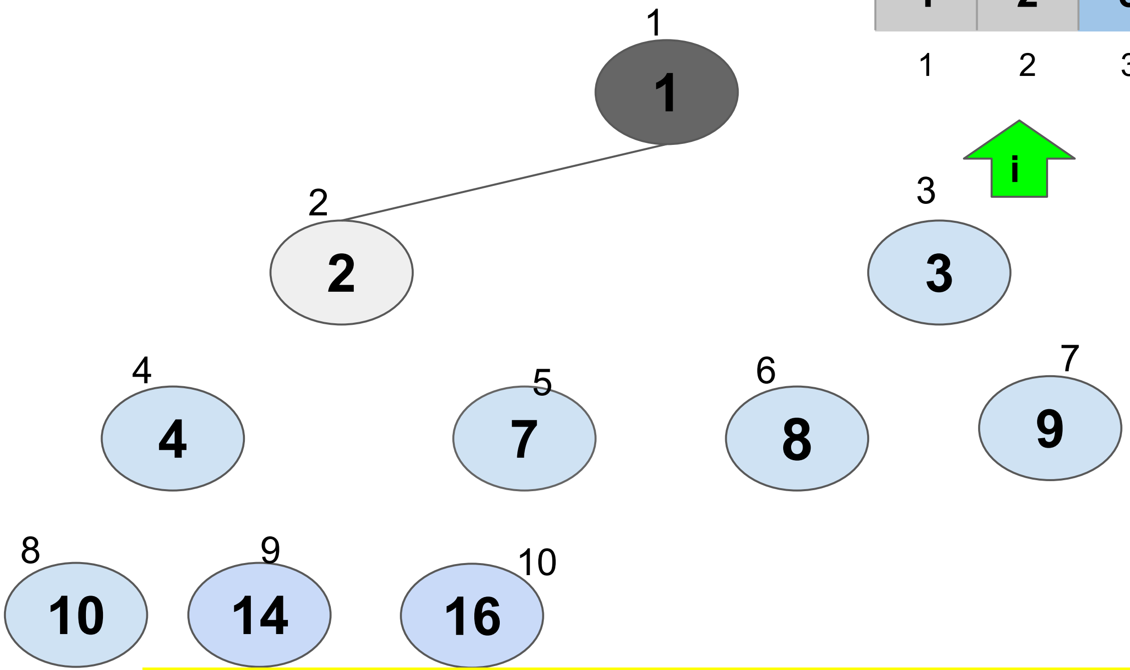
3	2	1	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10



- **Após a chamada Max-Heapify(A,1,3)**

Operação do HeapSort

1	2	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

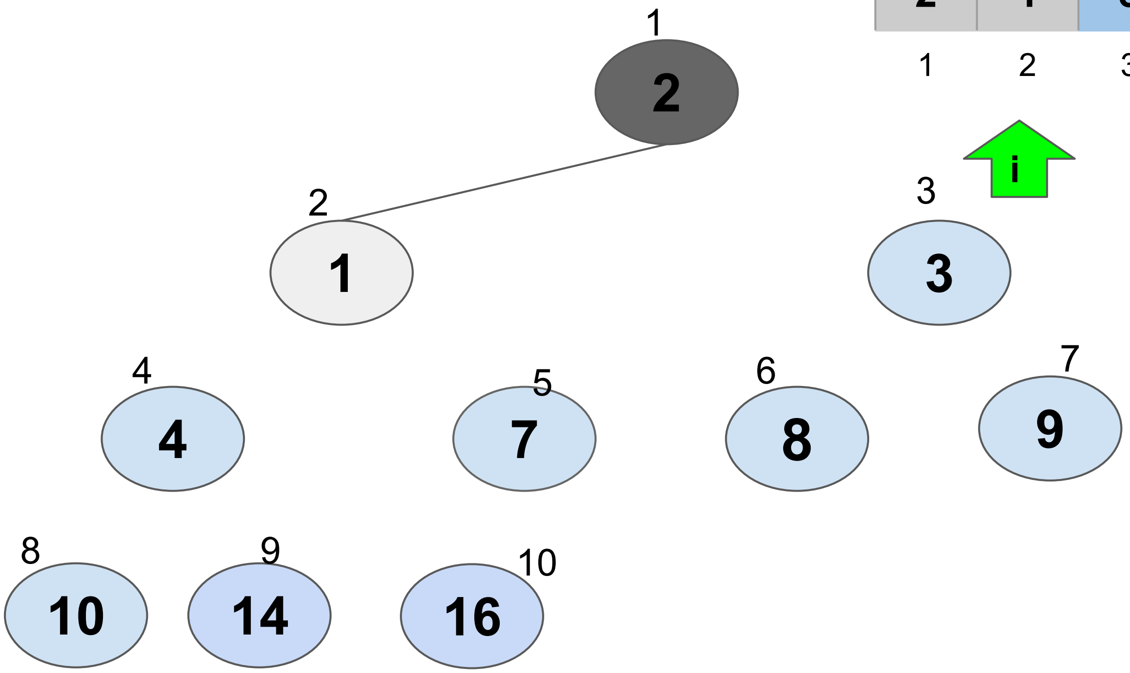


- **Após a chamada Max-Heapify(A,1,3)**
- **Laço aponta para o nó 3. Troca A[1] com A[3].**
- **Chama Max-Heapify(A,1,2)**

Operação do HeapSort

2	1	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

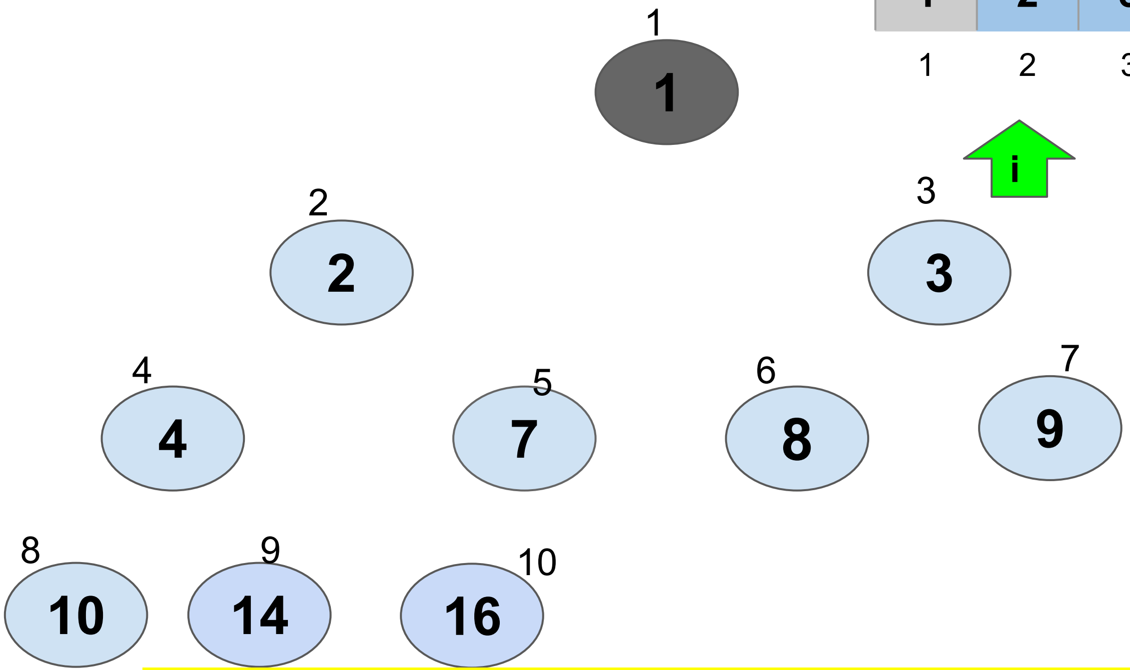
i



- **Após a chamada Max-Heapify(A,1,2)**

Operação do HeapSort

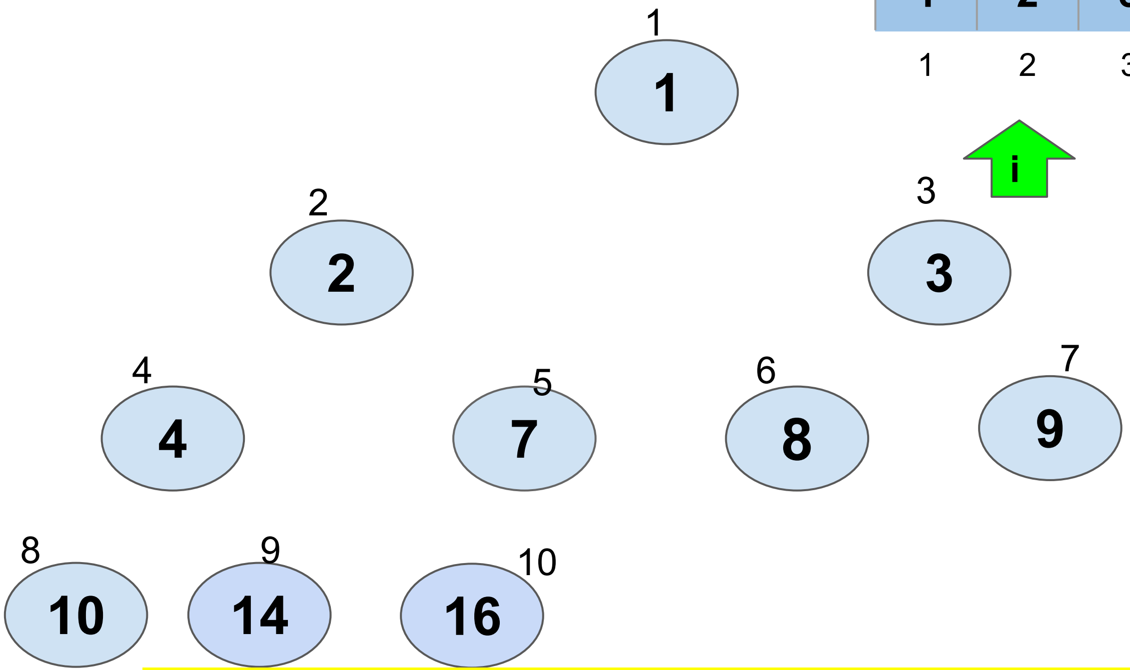
1	**2**	**3**	**4**	**7**	**8**	**9**	**10**	**14**	**16**
1	2	3	4	5	6	7	8	9	10



- **Após a chamada Max-Heapify(A,1,2)**
- **Laço aponta para o nó 2. Troca A[1] com A[2].**
- **Chama Max-Heapify(A,1,1)**

Operação do HeapSort

1	2	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10



- Chamada Max-Heapify(A,1,1) não casa efeito na estrutura
- Laço i finaliza
- Fim do HeapSort

HeapSort

- **Vantagens:**
    - O comportamento do Heapsort é sempre O(n log n), qualquer que seja a entrada.
- **Desvantagens:**
    - O anel interno do algoritmo é bastante complexo se comparado com o do Quicksort.
    - O Heapsort **não é estável**.
- **Recomendado**:
    - Para aplicações que não podem tolerar eventualmente um caso desfavorável.
    - Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o heap.

Comparação entre os métodos

Complexidade:

Algoritmo	Pior Caso	Caso Médio
Seleção	$O(n^2)$	$O(n^2)$
Inserção	$O(n^2)$	$O(n^2)$
MergeSort	$O(n \log n)$	$O(n \log n)$
QuickSort	$O(n^2)$	$O(n \log n)$
HeapSort	$O(n \log n)$	--

Comparação entre os métodos

Tempo de execução

	500	5000	10000	30000
Seleção	16.2	124	161	--
Inserção	11.3	87	228	--
Quicksort	1	1	1	1
HeapSort	1.5	1.6	1.6	1.6

Observação:

- O método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele.
- Registros na ordem aleatória

Exercício

1. Mostre a execução do `quicksort` no arranjo A abaixo usando como o pivô a mediana dentre três elementos.

   A = {40, 37, 95, 42, 23, 51,27}

2. Mostre a ação da rotina `Max-Heapify`(S,1,8) no arranjo

   A = { 95, 98, 78, 39, 28, 70, 33}

Referências

- Parte do material foi baseada nos slides dos professores:
  - Nivio Ziviani, Ph.D (DCC/UFMG)
  - Jairo Francisco de Souza (DCC/UFJF)
- Algoritmos: Teoria e Prática. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. Elsevier, 2012.